

# SmartQA Wisdom

**on**

**TEST DESIGN**

**Thiruvengadam Ashok**

STAG Software

*Wisdom on Test Design* delves into the deeper principles of intelligent testing, focusing on strategies that ensure high-quality software delivery. It encourages a mindset shift from simply executing tests to thoughtfully crafting them, with an emphasis on designing for testability and aligning testing practices with the broader goals of software development.

**Key topics include:**

- Developing a test design mindset, understanding system behavior, and creating impactful test scenarios.
- Prioritising testability with principles like SOLID design and the shift-left approach.
- Approaching complex UI testing with behavior-driven methods and manageable entities.
- Expanding coverage beyond code to include entities, environments, personas, and workflows.
- Leveraging heuristics to uncover subtle edge cases and corner cases.

By presenting testing as a craft rooted in curiosity, rational analysis, and purposeful design, *Wisdom on Test Design* provides a transformative perspective for testers, developers, and quality professionals. It is a must-read for anyone seeking to go beyond the surface and uncover the profound essence of intelligent software assurance.

## **About the author**

Thiruvengadam Ashok is the CEO of STAG Software Private Limited & Co-Founder of Pivotrics, based in Bengaluru, India. Ashok has dedicated his career to the pursuit of quality assurance in software, continuously evolving his approaches to match the needs of modern systems. He is the creator of HyBIST, an innovative approach to SmartQA that has revolutionised how teams approach hypothesis-driven testing.

Ashok's professional life is deeply intertwined with his personal philosophy. A passionate ultra-distance runner and long-distance cyclist, he applies the principles of endurance and exploration to his work, constantly seeking out new ways to improve software quality. He is also an avid wordsmith, using his love of language to weave both poetry and technical innovation into his life's work.

He holds an M.S. in Computer Science from the Illinois Institute of Technology, a Bachelor's degree in Electronics and Communication Engineering from the College of Engineering, Guindy, and a Postgraduate Diploma in Environmental Law from the National Law School of India University, Bangalore. His life maxim—"Love what you do & Do only what you love"—is reflected in everything he undertakes, from professional projects to personal passions.

**Copyright © 2025, Thiruvengadam Ashok**

All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations used in reviews and certain other noncommercial uses permitted by copyright law.

**Disclaimer:**

The information contained here is for educational and informational purposes only. While every effort has been made to ensure the accuracy of the content, the author and publisher make no representations or warranties regarding the completeness, accuracy, or applicability of the information provided. The strategies and methodologies described are for informational purposes and should be adapted to individual circumstances as necessary.

**Trademarks:**

All product names, logos, and brands mentioned in this book are the property of their respective owners. Use of these names, logos, and brands does not imply endorsement.

HyBIST is the intellectual property of STAG Software Private Limited.

**Edition: First edition, 2025.**

# TABLE OF CONTENTS

<b>Mindset for Smart Test Design</b>	<b>7</b>
What does it take ?	7
Thinking styles	7
<b>Designing in Testability</b>	<b>8</b>
Introduction	8
Background of DFT	8
The economic value of DFT	8
Why is testability important?	9
'SOLID' design principles	9
Law of Demeter (LoD)	10
Guidelines to ease testability of codebase	10
<b>Test Design Approaches</b>	<b>11</b>
#1 User view based	11
#2 Analytical view based	11
#3 Construction view based	12
#4 Test view based	12
#5 Experience-view based	12
#6 Operational-view based	12
#7 Evolution-based	13
<b>Design Approach for Rich UI</b>	<b>14</b>
<b>Smart coverage framework</b>	<b>20</b>
Code coverage	20
Entity coverage	21
Environment coverage	21
Test coverage	21
Persona coverage	21
<b>Some Heuristics for Identifying Corner Cases</b>	<b>23</b>

#1 Heuristic based on TIME	23
#2 Heuristic based on LIFECYCLE	23
#3 Heuristic based on TRANSFORMATION	23
#4 Heuristic based on POSITION	24
#5 Heuristic based on SPACE	24
#6 Heuristic based on SIZE	24
#7 Heuristic based on LINKAGES	24
#8 Heuristic based on LIMIT	25

# SMARTQA WISDOM ON TEST DESIGN

## **MINDSET FOR SMART TEST DESIGN**

Testing design is often overshadowed by test execution and automation. However, the ultimate goal of testing is to deliver clean, high-quality code. Good test design is rooted in a deeper understanding of quality and functionality. To achieve this, test design should involve sensitivity to the system's behavior, focusing on the prevention and detection of defects through intelligent test scenarios. A thoughtful approach to test design ensures that the system is tested thoroughly while minimising redundancy and inefficiencies.

## **What does it take ?**

Designing tests with a smart mindset requires incorporating efficiency, simplicity, and clarity from the start. Prioritising testability, considering multi-perspective views, and focusing on intent enhances the design approach. Historical insights and simplifying complexity further ensure that tests are robust and relevant to real-world conditions.

## **Thinking styles**

Smart test design is about thinking critically, questioning deeply, and designing intelligently. It requires more than just writing test cases—it demands a mindset of efficiency, precision, and adaptability.

A shift-left mindset means engaging early, focusing on prevention over detection. A risk-driven perspective ensures effort is spent where it matters most, not in exhaustive, unfocused testing.

Systems thinking is key—seeing beyond isolated components to understand how interactions shape behavior. A hypothesis-driven mindset questions everything: What can go wrong? Where might it break? This fuels curiosity, the hallmark of great testers.

Tests should be minimalist yet comprehensive, designed with clarity and impact. Thinking in terms of real-world behavior rather than just system requirements ensures relevance. Automation awareness helps in making strategic decisions about what to automate and what to explore manually.

A smart tester is data-driven, always learning from patterns, failures, and evolving requirements. Above all, adaptability is crucial—testing is never static, and neither is the mindset behind it.

# SMARTQA WISDOM ON TEST DESIGN

## DESIGNING IN TESTABILITY

### Introduction

Software testability is the degree to which a software artefact (i.e., a software system, software module, requirements, or design document) supports testing in a given test context. If the testability of the software artefact is high, identifying faults in the system (if any) through testing becomes easier.

The correlation between 'testability' and good design is evident in the observation that code with weak cohesion, tight coupling, redundancy, and a lack of encapsulation is difficult to test. A lower degree of testability results in increased test effort. In extreme cases, a lack of testability may even hinder testing certain parts of the software or its requirements altogether. (From [1] "Software testability")

Testability is a product of effective communication between development, product, and testing teams. The more testability is considered during feature creation, and the more testers are involved during this phase, the more effective the testing process becomes. (From [2] "Knowledge is power when it comes to software testability")

### Background of DFT

Design for testability (DFT) is not a new concept. It has been applied in electronic hardware design for over 50 years. If an integrated circuit needs to be testable both during its design stage and in production, it must be designed with testing in mind. These "hooks" for testability must be embedded at the design phase; they cannot be added later, as the circuit is already in silicon and cannot be changed.

DFT is a critical non-functional requirement influencing nearly every aspect of electronic hardware design. Similarly, complex agile software systems require testability considerations during both the design and production phases. Without designing software for testability, testing it effectively post-development becomes infeasible.

(From [3] "Design for testability: A vital aspect of the system architect role in SAFe")

### The economic value of DFT

Agile testing addresses two specific business objectives:

1. Critiquing the product by minimising the impact of defects delivered to users.
2. Supporting iterative development by providing quick feedback within a continuous integration process.

These goals are challenging to achieve if the system does not support simple system, component, or unit-level testing. Agile programmes that emphasise testability through every design decision enable the enterprise to achieve shorter runways for business and architectural epics. DFT reduces the impact of large system scope and allows agile teams to



## SMARTQA WISDOM ON TEST DESIGN

work with more manageable, high-quality assets, minimising development delays and reducing the need for rework.

(From [3] "Design for testability: A vital aspect of the system architect role in SAFe")

### **Why is testability important?**

Testability impacts deliverability. When testers can easily locate issues, debugging is faster, and the application reaches the user more quickly and with fewer hidden glitches. Higher testability provides product and development teams with faster feedback, enabling frequent fixes and iterations.

**Shift-left:** Rather than waiting until the testing phase, adopting a whole-team approach to testability means considering it thoughtfully during planning, design, and development. This includes focusing on key aspects such as documentation, logging, and requirements. The more knowledge testers have about the product or feature—its purpose and expected behaviour—the more valuable their testing efforts and results will be.

(From [2] "Knowledge is power when it comes to software testability")

**Exhaustive testing** is more practical and achievable when applied in isolation for each component on all possible measures. This approach enhances quality, rather than attempting to test the finished product through use cases that address all components simultaneously.

This raises the question: "Are all components testable?" The answer is to design components to be as highly testable as possible. Additionally, optimal system-level tests should complement isolated tests to ensure end-to-end completeness. Exhaustive testing involves placing the right set of tests at the appropriate levels, balancing isolated and system-level tests effectively.

(From [4] "Designing the software testability")

### **'SOLID' design principles**

These principles can help you write easily testable code that is not only more flexible and maintainable but also better modularised:

- 1. Single responsibility principle (SRP):** Each software module should have only one reason to change.
- 2. Open/closed principle (OCP):** Classes should be open for extension but closed to modification.
- 3. Liskov substitution principle (LSP):** Objects of a superclass should be replaceable with objects of its subclasses without breaking the application.
- 4. Interface segregation principle (ISP):** No client should be forced to depend on methods it does not use.

## SMARTQA WISDOM ON TEST DESIGN

- 5. Dependency inversion principle (DIP):** High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details, but details should depend on abstractions.

(SOLID = SRP + OCP + LSP + ISP + DIP)

(From [5] "Writing testable code")

### Law of Demeter (LoD)

This principle helps maintain decoupling and testability by stating that:

- Each unit should have limited knowledge about other units, interacting only with closely related ones.
- Each unit should only communicate with its "friends" (immediate dependencies) and avoid interacting with "strangers" (distant dependencies).

(From [5] "Writing testable code")

### Guidelines to ease testability of codebase

- 1. Make sure your code has seams:** Seams allow behaviour in your programme to be altered without editing its core logic.
- 2. Separate object creation from application logic:** Use factories for object creation, keeping application classes focused on business logic.
- 3. Use dependency injection:** Avoid creating or fetching dependencies within a class. Provide dependencies through constructors or other mechanisms.
- 4. Avoid global state:** Global state complicates code understanding and increases the potential for test flakiness.
- 5. Avoid static methods:** Static methods lack the seams needed for unit testing and should be avoided in object-oriented paradigms.
- 6. Favour composition over inheritance:** Composition promotes modularity, making the code easier to test and avoiding the proliferation of complex class hierarchies.

(From [5] "Writing testable code")

### References

[1] Software testability at [https://en.wikipedia.org/wiki/Software\\_testability](https://en.wikipedia.org/wiki/Software_testability)

[2] "Knowledge is power when it comes to software testability" <https://smartbear.com/blog/test-and-monitor/knowledge-is-power-when-it-comes-to-software-testa/>

[3] "Design for testability: A vital aspect of the system architect role in SAlFe" at <https://www.scaledagileframework.com/design-for-testability-a-vital-aspect-of-the-system-architect-role-in-safe>

[4] "Designing the software testability" at <https://medium.com/testengineering/designing-the->

# SMARTQA WISDOM ON TEST DESIGN

software-testability-2ef03c983955

[5] "Writing testable code" <https://medium.com/feedzaitech/writing-testable-code-b3201d4538eb>

## TEST DESIGN APPROACHES

Test design often emphasises execution and the ability to cover more ground. The focus has shifted towards how frequently tests can be executed, often prioritising automation. However, it is essential to revisit the primary objective, which is to deliver clean code. This requires a deeper sensitivity to the quality of tests, highlighting the importance of thoughtful test design.

A smart approach to test design involves creating effective scenarios—ideally fewer in number—that can uncover the most critical issues.

Smart Test Design is about looking at the system from multiple views to:

**I want | expect | would-like behaviours to satisfy needs that are implemented well and comprehensively covered to help me do well on my environments with no side effects.**

Decide what to prevent, detect statically, or test—whether through human effort or automation. Prioritise intent first, followed by the corresponding activity.

Here are TWENTY approaches to smart test design, viewed through SEVEN distinct views.

### #1 User view based

- 1.1 Use the requirement specification to design.
- 1.2 See actual users of how they work and then use this to design.
- 1.3 With users doing experience sessions with system and use this information to design.



### #2 Analytical view based

- 2.1 Use software test techniques (black or white) on spec/structure to design.
- 2.2 Construct behavior models and use this to design.
- 2.3 If the system needs to comply with a standard, use the standards information to design.



# SMARTQA WISDOM ON TEST DESIGN

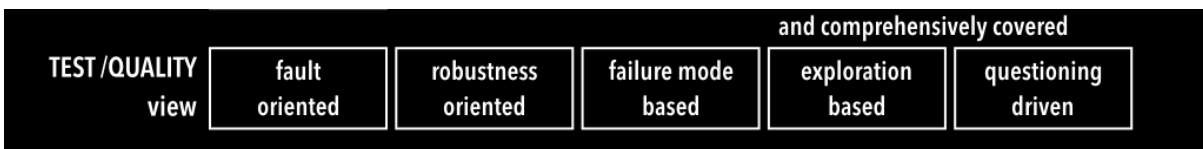
## #3 Construction view based

- 3.1 Use the code properties like lines/conditions/path to design.
- 3.2 Exploit your deep understand of technology to understand potential mechanisms/flaws and use this design.
- 3.2 Understand how the system has been architected, composed & integrated to design.



## #4 Test view based

- 4.1 Hypothesise potential faults probable given your understand of usage, structure, architecture, environment, conditions to design.
- 4.2 Use potential error return codes, exceptions, deliberate bad inputs, violations of system states to design.
- 4.3 Identify potential end failures and failure modes and use this to design.
- 4.4 Explore the system to understand its behaviour in various contexts and use this to design .
- 4.5 Probe the system with a series of questions (say what-if) and use this to design.



## #5 Experience-view based

- 5.1 Use the past history of issues encountered with various customers and design.
- 5.2 Apply the learning of various situations or deeper knowledge to come u with fault patterns and use this to design.



## #6 Operational-view based

- 6.1 Use the understanding of actual business flows, usage profiles of features on various environments to design.
- 6.2 Identify various deployment configurations and use this information to design.



# SMARTQA WISDOM ON TEST DESIGN

## #7 Evolution-based

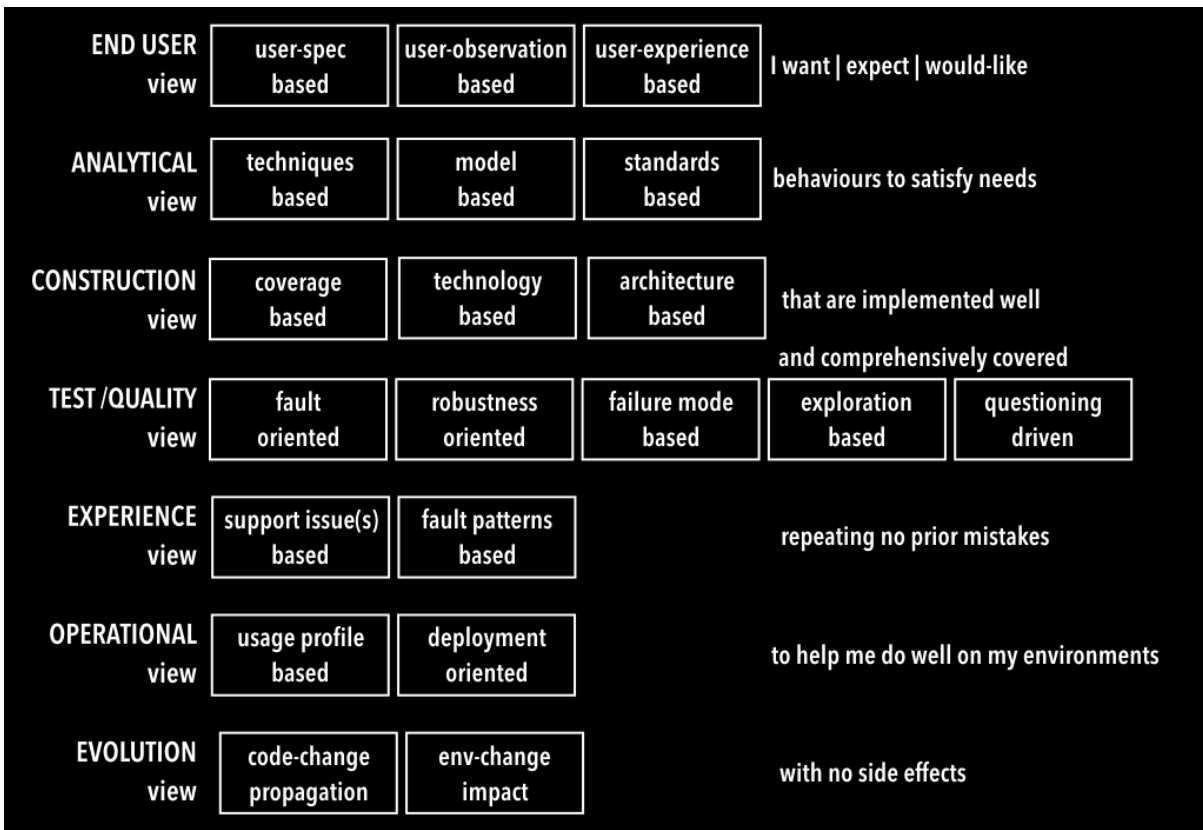
7.1 Use the information of what code has been changed, to understand how these may propagate and may cause issues to design.

7.2 Use the information of changes in the environment to understand their impact on the system and design



Summarising Smart Design is about:

I want | expect | would-like behaviours to satisfy needs that are implemented well and comprehensively covered to help me do well on my environments with no side effects.



### SMART DESIGN

*"I want | expect | would-like behaviours to satisfy needs that are implemented well and comprehensively covered to help me do well on my environments with no side effects"*

# SMARTQA WISDOM ON TEST DESIGN

## DESIGN APPROACH FOR RICH UI

During a consulting engagement, a tester presented a complex decision table to represent a behaviour model. Upon closer examination, the behavioural conditions were found to have been derived by visually examining the complex UI, rather than delving deeper to understand the underlying behaviour. This challenge with behaviour modelling is not uncommon and has been observed in numerous organisations.

The question arises: why does a complex UI overwhelm testers? The likely reason is an inability to see beyond the UI and to question beyond the obvious. The outcome is often an excessive number of test cases and an overwhelming perception that scientific modelling and design cannot be applied due to time constraints. However, this is where the intellectual challenge of testing comes into play—decomposing an entity through rational analysis and curiosity to design smarter tests.

A rich UI typically contains numerous data fields, some of which depend on others or system settings. These fields may be organised across multiple tabbed dialogues or spread across different screens, with certain features appearing in multiple locations. This complexity creates challenges, including determining where to begin modelling the behaviour and managing mixed test cases related to fields, UI interactions, behaviour, business flows, and environmental dependencies.

Let us examine a problem to understand how testing a rich UI can be simplified, using an airline application structured across four screens, each with its own complexities.

(From [Awery Software Screenshots](#))

### 1. Setting up a flight schedule for a flight

Flight ID	Date	Time	Origin	Status	Destination	Time
100710001	10/07/2010	15:00	HDAM	planned	OMDB	23:50
XX-1234	A300-600F	flight	111	15:00	23:00	
100709001	27/07/2010	02:40	OKBK	planned	OMSJ	00:00 - 00:00
1234	A300-600F	flight	1123	02:35	17:40 - 00:00	OMDB 17:40 00:00
DEMO Customer	555667	(null)	L C A	(null)	xxx-111	15 12345 FH L
10728003	28/07/2010	00:00	OMDB	planned	UAFM	00:00
-1234	A300-600F	flight	xx-1111	00:00	00:00	
100728001	28/07/2010	14:50	OMDB	planned	OKBK	15:40 - 17:30
AA-3676	A300-B4	flight	XX-1211	14:50	15:40 - 16:20	UAFM 18:50 - 19:30 OMSJ 23:40
DEMO Customer	XX-1211	F	H		00:00 - 00:00	00:00
100728002	28/07/2010	15:00	OMSJ	planned	OKBK	00:00 - 00:00
AA-3676	A300-B4	flight	XX-1211	00:00	00:00 - 00:00	OMDB 00:00
DEMO Customer	XX-1211	(null)	(null)	(null)	(null)	(null)

# SMARTQA WISDOM ON TEST DESIGN

## 2. Setting the route information - "start location details"

Flight No.: 100709001 Status: planned Type: Flight

Waypoint: **Route: OKBK** 1 Call sign: 1123

Departure: 27/07/2010 02:40

Status: planned Permission Timing: 27/07/2010 00:00

**VIA Details**

<input checked="" type="checkbox"/> Fueling	Place/Note:	QTY: 5000	Payment: Credit Cash	Vendor: Fuel company 1
<input type="checkbox"/> Handling	Place/Note:		Payment: Credit Cash	Vendor: Vendor 1
<input checked="" type="checkbox"/> Landing	Place/Note:		Payment: Credit Cash	Vendor: Avendor
<input checked="" type="checkbox"/> Catering	Place/Note:	QTY: 0	Payment: Credit Cash	Vendor: Dvendor
<input type="checkbox"/> Hotel	Place/Note:	QTY:	Payment: Credit Cash	Vendor: None

## 3. Setting up the route information - "permissions"

Flight No.: 100709001 Status: planned Type: Flight

Waypoint permissions (0): **Perm: OMDB** Pre permissions (0): Block permissions (0):

Name	Perm no	Sent	Approved	Denied	Canceled	Call sign	From	Till	Note
LNDG PRMT	1234					xxx-111			

**AP perms**

LNDG PRMT	24
NEW FLT CREATION	12 HR

Name: LNDG PRMT Call sign: xxx-111

Perm No: 1234 Note:

From date: 08/10/2010 17:00  Sent

To date: 08/10/2010 18:00  Approved

Sent file: LOADING 0%  Canceled

Recived file: LOADING 0%  Denied

E-mail 1: E-mail 2: E-mail 3: E-mail 4: Tel: +97142161600 Fax: +97142162272 SITA: DXBAPYF AFTN: DSN: Note: Send: E-mail Sita AFTN From: Blank te...

# SMARTQA WISDOM ON TEST DESIGN

## 3. Setting up the route information - "cargo details"

Home page Flights 100709001 OKB... 100709001 OMDB

Refresh Save AC: A300-600F Regno: XX-1234 Flight No.: 100709001 Status: planned Type: Flight Flights

OKBK planned 02:40 OMSJ planned 00:00 - 00:00 OMDB planned 17:40  
02:35 17:40 - 00:00 00:00  
1123 555667 F L C (null) 15 12345 F H L

Route: OMDB Perm: OMDB Block Perm Fuel Charter **Cargo** Plan Tracking Crew Corr

Load plan: OKBK to: OMDB GET GET XLS GET FULL GET FULL XLS With AMD

AWBs Cargo shipments Load Pieces in OMDB Unload Pieces in OMDB Transit Pieces in OMDB Takeoff Pieces in OMDB

Refresh AWB No.: Orig None Dest None Customer Total AWBs/AMDs: 6 Total pieces: 46 Total weight: 1963.7 Total volume: 18.77

Quick search:

	AWB No.	Status	Palette	Part No	Origin	Dest	Customer	PCS	Weight	Volume	Prior	D
<input type="checkbox"/>	041-00015856	DEPARTURE					BAXMIL	20	200	2.00	medium	28/07/2
<input type="checkbox"/>	DBSC0185012	DEPARTURE			OKBK	HDAM	BAXMIL	3	101.41	0.00	1	07/07/2
<input type="checkbox"/>	DBSC0185014	DEPARTURE				DDKS		1	88.18	0.00	0	07/07/2
<input type="checkbox"/>	DBSC0185028	NOT-COMPLETE				ORBD		3	432.11	0.00	0	07/07/2
<input type="checkbox"/>	111-17116326	NOT-COMPLETE			OMDB	ORBI	DEMO Cust	17	940	16.77	high	29/05/2
<input type="checkbox"/>	111-61077181	DEPARTURE			OMDB	ORSU		2	202	0.00	low	06/05/2

## 4. Setting up the route information - "setting up crew"

Home page Flights 100709001 OKB... 100709001 OMDB

Refresh Save AC: A300-600F Regno: XX-1234 Flight No.: 100709001 Status: planned Type: Flight Flights

OKBK planned 02:40 OMSJ planned 00:00 - 00:00 OMDB planned 17:40  
02:35 17:40 - 00:00 00:00  
1123 555667 F L C (null) 15 12345 F H L

Route: OMDB Perm: OMDB Block Perm Fuel Charter Cargo Plan Tracking **Crew** Corr

Flight Crews Flight Passengers

Refresh Crews Base GD From: OKBI To: OKBI Get Selected Now: From: OKBK To: OMDB assign to this flight

Available positions: CAPTAIN FIRST OFFICER (co-pilot) FLIGHT NAVIGATOR LOAD MASTER

All by position. All Last crew Avail crew for flight

Available by sel. posit.: Search:

Name: Adam Jones Smith Mark Name Test surname Selected: Paul Zidan CAPTAIN

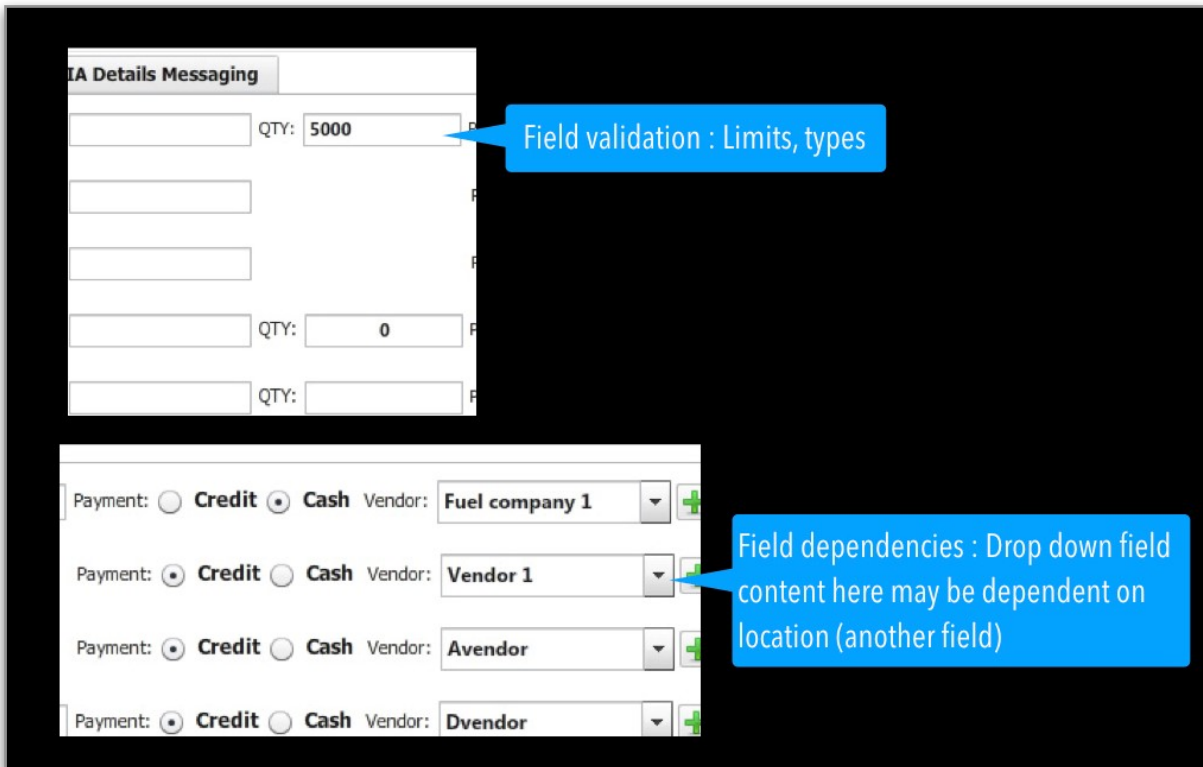
Name	Position	From	To
Mark Smith	CAPTAIN	OKBK	OMDB
Ivanov Nikolaj	FIRST OFFICER (	OKBK	OMDB

Let's analyse what is be observed in these rich UI screens.

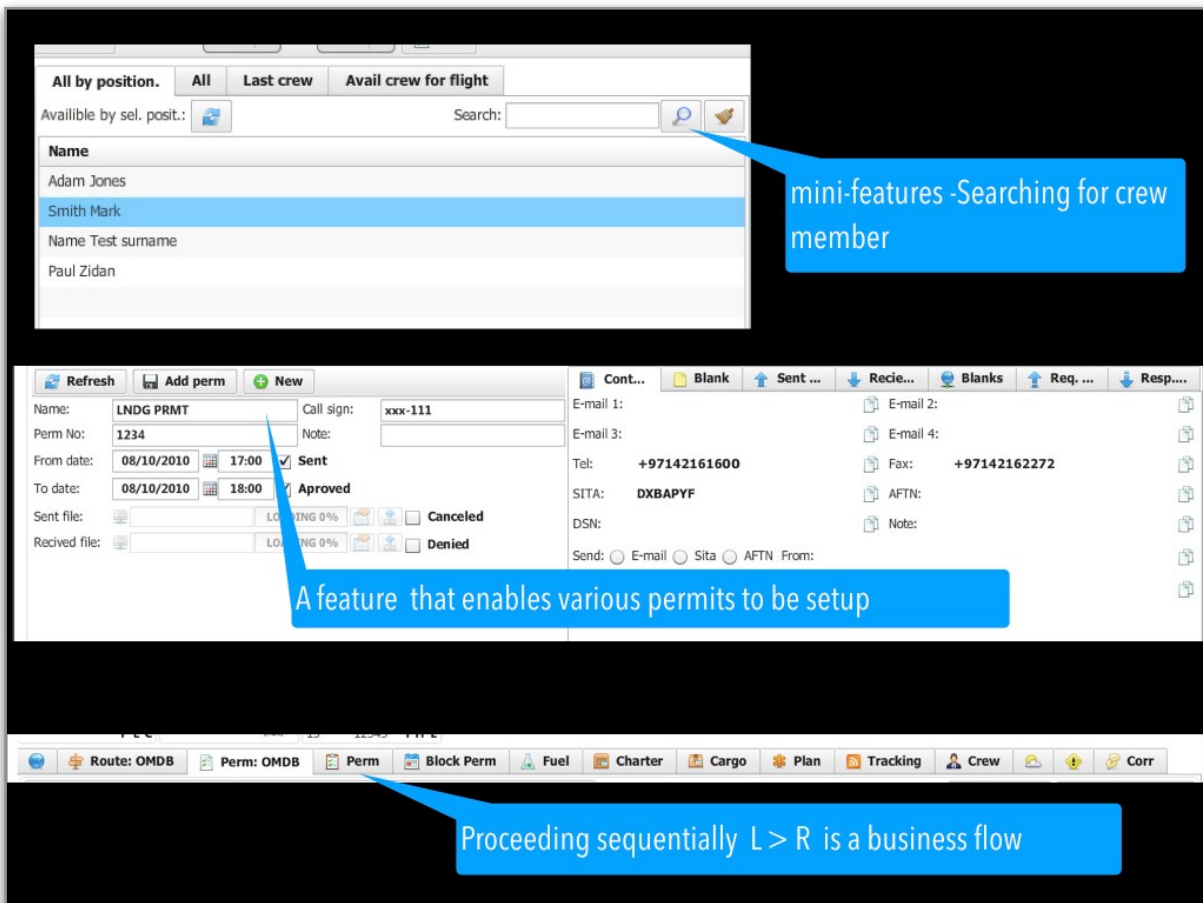
1. Firstly, there are data fields—both independent and dependent.



# SMARTQA WISDOM ON TEST DESIGN

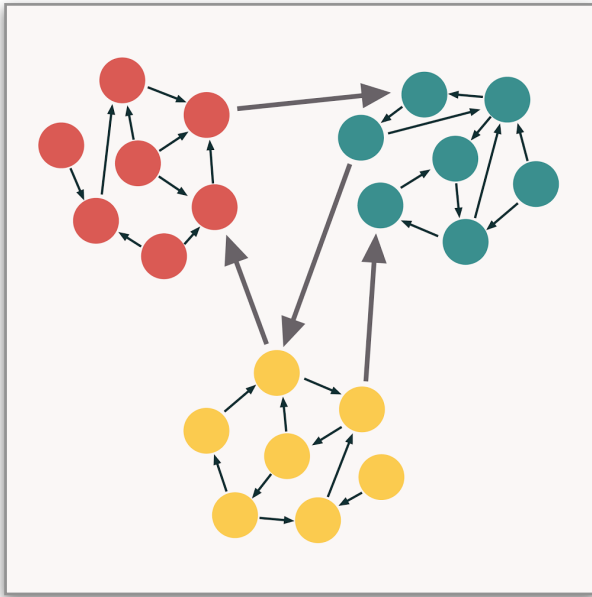


2. Secondly there are features/functions (some of them a mini feature) and a business flow that combines multiple features.



## SMARTQA WISDOM ON TEST DESIGN

So, this single rich UI screen consists of a set of features, and enables a complex flow to be done. How do we test this?



Let us look at good design principles "COHESION & COUPLING ". Good design of software is high cohesion and loose coupling. Grouping similar actions & data (cohesion) keeping minimal dependencies between them (coupling).

How is this useful in TEST DESIGN for rich UI?

**Landing permit feature**

**Cargo setup feature**

**Crew assignment feature**

See the UI as a set of highly cohesive 'clumps' of fields that form a feature

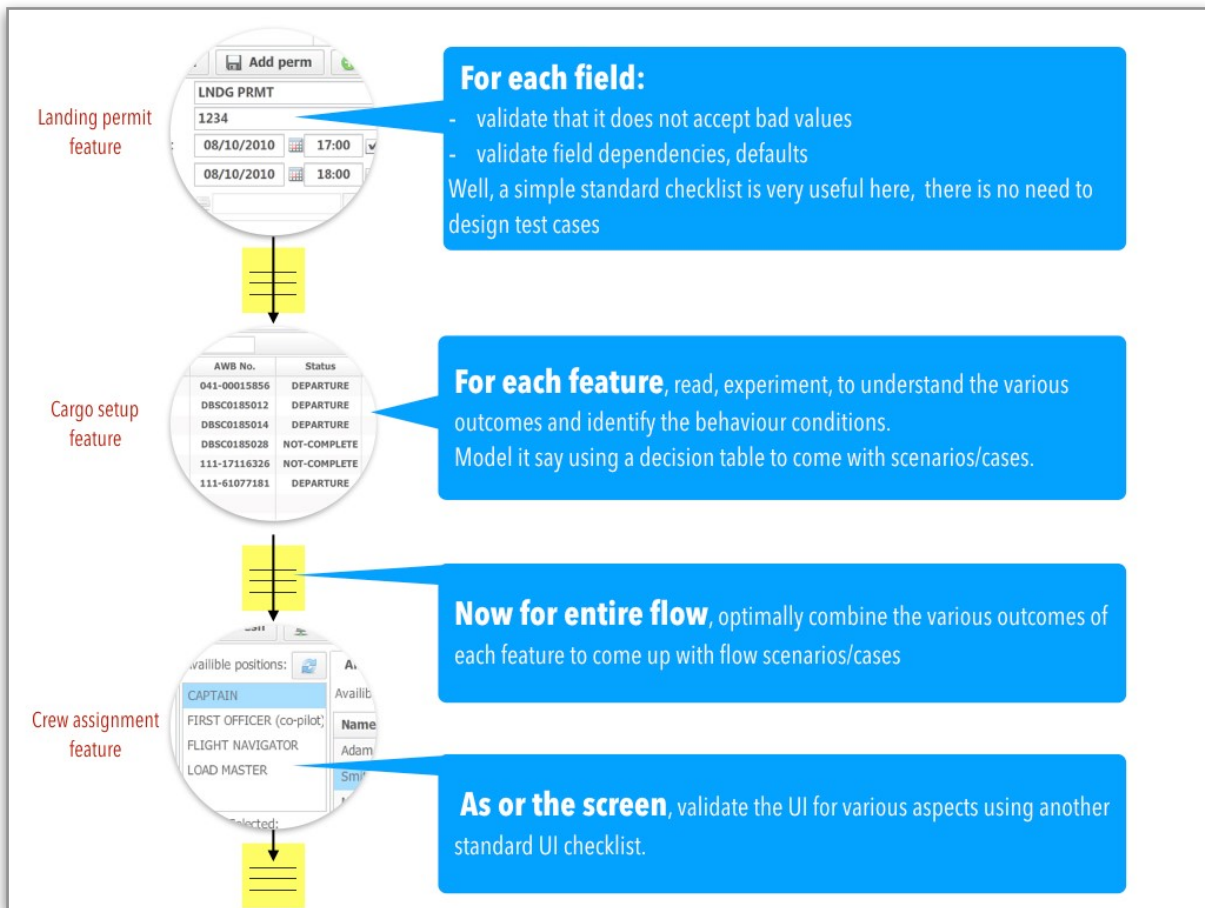
See the UI as a set of interconnected features that form a business flow

Now be clear of what you want to design for : field, feature, flow

Now figure out what issue you want to uncover in field/feature/flow and then design test cases

Let us dig in...

## SMARTQA WISDOM ON TEST DESIGN



So, what have we done to tackle challenges of complex UI? We have:

1. DECOUPLED complex UI into COHESIVE entities : FIELD, FEATURE, FLOW & UI
2. Applied behavior driven approach for feature test design (decision table)
3. Combined outputs (variations) optimally to test the FLOW (variations table)
4. Used checklist as needed to leverage prior design

In a rich UI, one tends to see HOW-to-do, whereas if it was UI-less, one seems to focus on "WHAT-is-to-be-done.

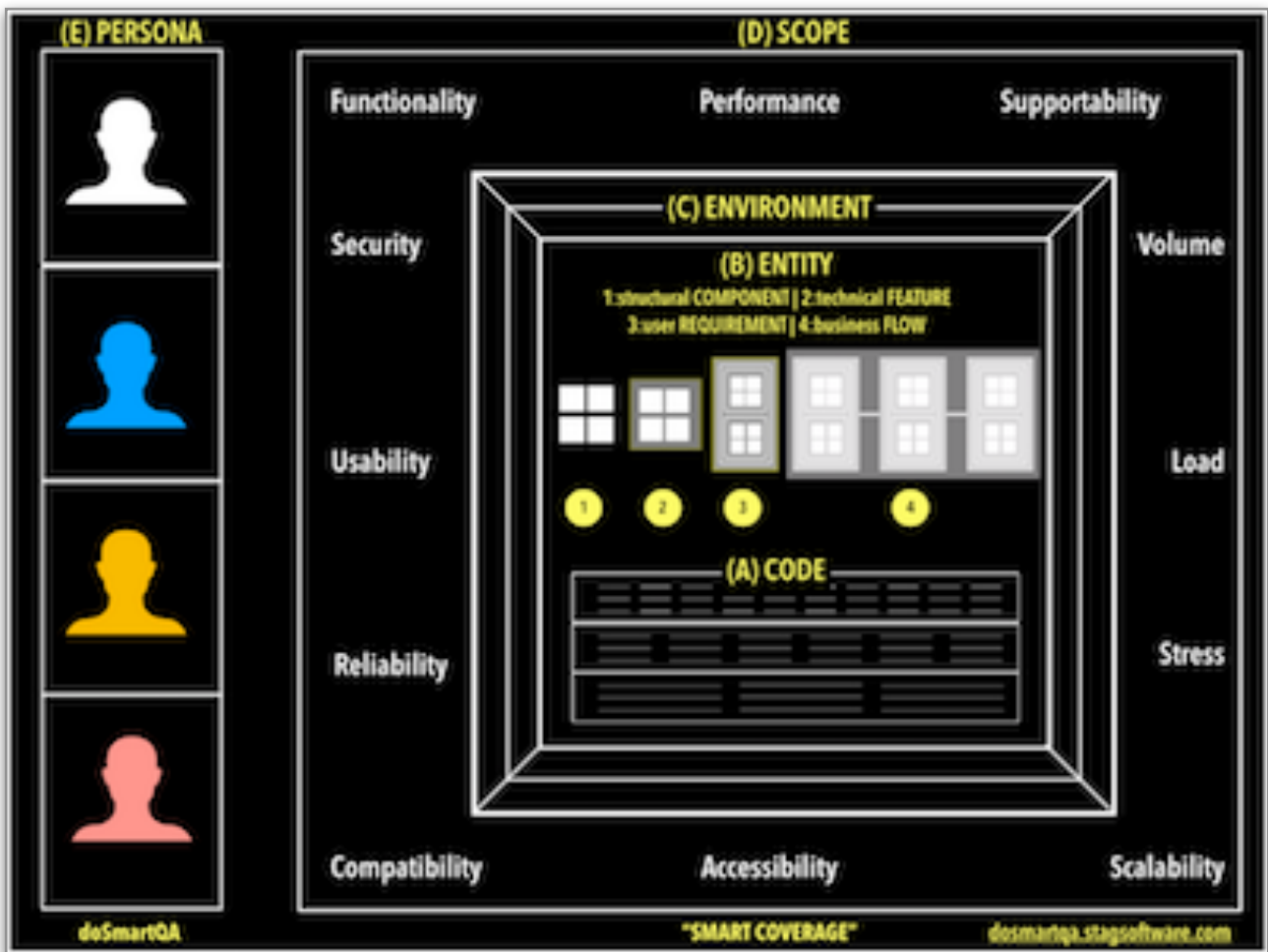
The latter is important for a tester to figure out 'WHAT-ifs'.

# SMARTQA WISDOM ON TEST DESIGN

## SMART COVERAGE FRAMEWORK

We are often challenged on the adequacy of test cases. The traditional concept of code coverage, while necessary, remains a limited and one-dimensional metric. Although it serves as a baseline, it is far from sufficient. Other coverage notions, such as Requirements Traceability Matrix (RTM) and test coverage, have yet to be rigorously developed to support objective evaluations.

To address the need for rapid and comprehensive assessment, we propose a simple yet holistic smart coverage framework that provides a complete 360-degree perspective."



### Code coverage

"At a structural level, code coverage measures whether all code elements—such as lines, conditions, paths, functions, classes, and files—have been executed at least once. This helps validate that the implemented code has been exercised through functional test cases, serving as an indicator of the completeness of these tests, particularly in the early stages of testing.

However, it's important to recognise that code coverage only assesses the execution of existing code; it does not detect missing code or unimplemented functionality."

## SMARTQA WISDOM ON TEST DESIGN

### **Entity coverage**

At a behavioural level, entity coverage assesses whether all entities—ranging from lower-level components to higher-level flows—have been validated through appropriate test cases. A system is composed of various interconnected entities, and it is essential to ensure that each entity, whether at the granular level (e.g., components or features) or at the broader level (e.g., requirements or flows), has been thoroughly validated.

This approach ensures a comprehensive perspective, addressing both 'in-the-small' (individual components) and 'in-the-large' (integrated flows), thereby confirming that all aspects of the system have been effectively validated.

### **Environment coverage**

Have we validated the system across all relevant environments? An environment comprises various elements such as the operating system, browser, database, hardware devices, and more. Environment coverage ensures that validation has been performed across all critical combinations of these elements to guarantee compatibility and reliability in real-world scenarios.

### **Test coverage**

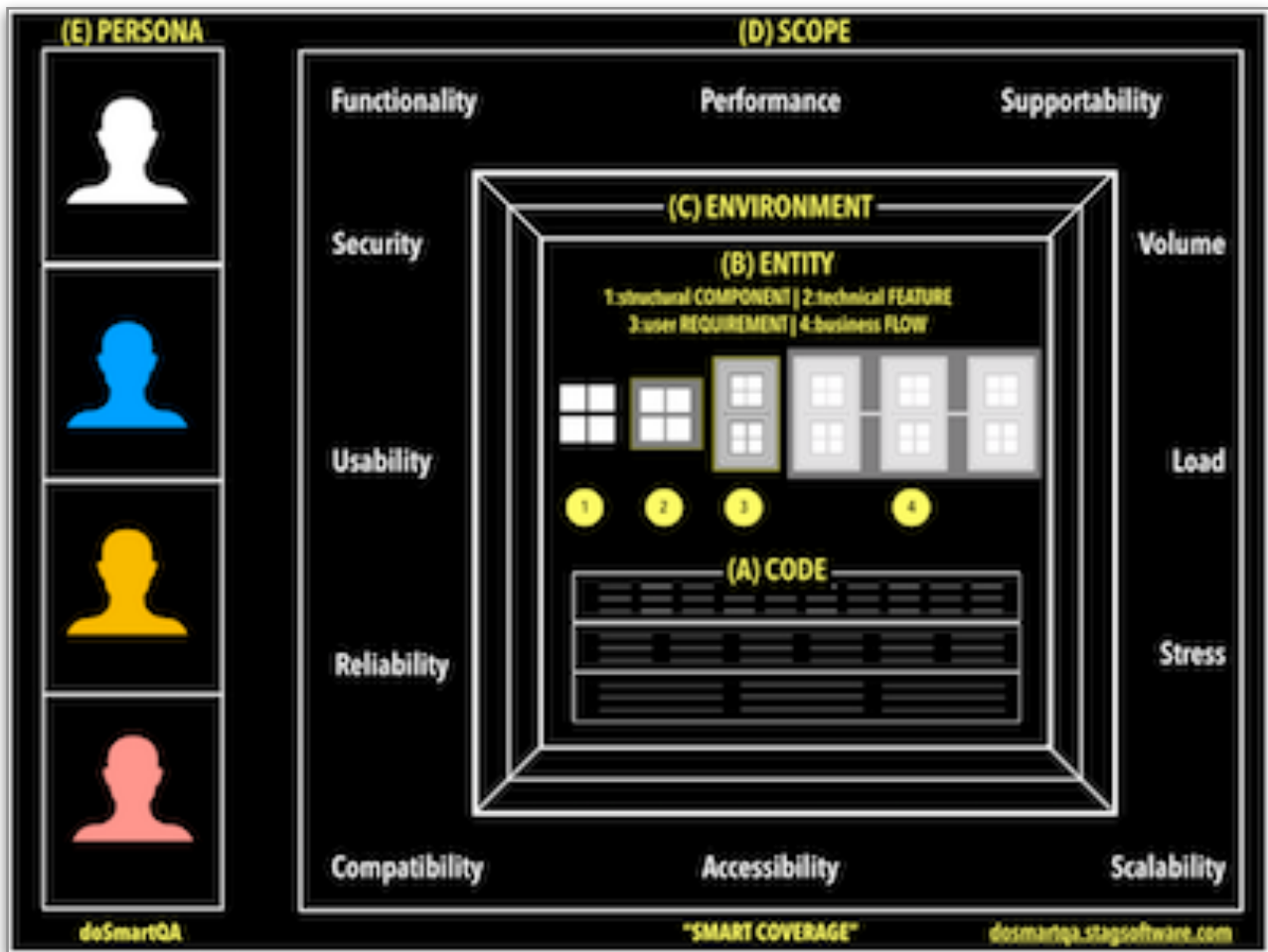
Have we designed test cases that encompass a variety of test types to evaluate the full scope of the system? The scope includes both functional correctness (functionality) and non-functional attributes (performance, security, etc.). Achieving comprehensive test coverage ensures that all relevant types of tests are executed, addressing both functional and non-functional requirements effectively.

### **Persona coverage**

Finally, have we considered scenarios from a persona perspective to ensure correctness from the user's point of view? After validating the product across dimensions like code, environment, entities, and test types, the final step is to evaluate it through the lens of the end consumer. Persona coverage ensures the system has been validated from the perspective of various personas (or actors), recognising that not all end users may be human. This approach helps align the system's functionality and usability with the needs of its intended users.

## SMARTQA WISDOM ON TEST DESIGN

*Concluding coverage should not be viewed as a metric applied only after the design phase; instead, it should serve as a guiding principle driving rapid and effective design from the outset.*



## SMARTQA WISDOM ON TEST DESIGN

### SOME HEURISTICS FOR IDENTIFYING CORNER CASES

As developers, we often focus on solving problems for typical or generic cases, which can lead to overlooking critical edge cases. For instance, while we may handle a system's normal operations effectively, we might miss scenarios like its behavior during the first-time setup.

Here I outline EIGHT heuristics that I discovered while testing a SaaS platform we were building. These heuristics are designed to uncover potential issues and are based on the following dimensions: Time, Lifecycle, Transformation, Position, Space, Size, Linkages, and Limit.

#### #1 Heuristic based on TIME



The first use of a system, such as creating a project, registering a user, performing an initial transaction, or purging content, highlights behaviours unique to a fresh or final state. These actions reflect transitions from start to end within the system.

#### #2 Heuristic based on LIFECYCLE



Repetition of system states involves cycling through starting, performing activities, and reaching an end, then restarting and continuing. A workflow may remain half-done, be suspended and resumed, or conclude through abandonment or logical closure.

#### #3 Heuristic based on TRANSFORMATION



The notion of transformation involves changes like formats or views, as seen in UI responsiveness at extreme view sizes. For content, it includes transformations to extremes, such as overly large, small, or null values.

## SMARTQA WISDOM ON TEST DESIGN

### #4 Heuristic based on POSITION



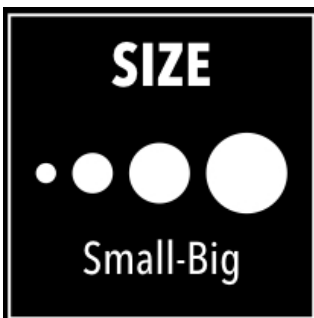
Observe behaviours of elements at the start or end, and how those in the middle behave when shifted to either end.

### #5 Heuristic based on SPACE



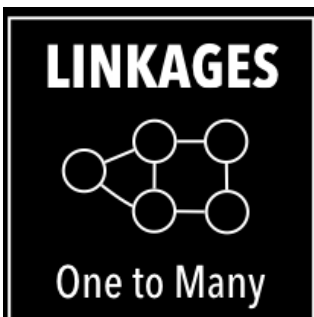
The notion of space involves contents being close or far, shrunk or expanded, particularly at extremes of proximity or size. An example is a responsive UI adjusting as the screen shrinks or expands.

### #6 Heuristic based on SIZE



The notion of volume or size includes extremes, such as uploading very large files or extremely small ones. For display, it could involve showing tiny or large content, perhaps through zoom.

### #7 Heuristic based on LINKAGES



The notion of linkages includes patterns like 1-1, 1-N, or N-N, with increasing chain lengths such as 1-1-1 or N-1-N. This examines linkage integrity, especially when  $N=0$  or chains propagate with varying  $N$ .



#8 Heuristic based on LIMIT



Extremes of value, such as minimum and maximum, are the most commonly understood within a defined range.



*"We are SmartQA evangelists. For over two decades we have transformed how individuals, teams and organisations have practised testing. We espouse methodology to test intelligently. Our mission - Elevate to high performance via SmartQA."*  
[www.stagsoftware.com](http://www.stagsoftware.com)



### The HyBIST Approach to SmartQA - MASTERCLASS

Testing is deep probing to seek clarity and in the process uncover, preempt issues rapidly. The HyBIST approach enables designing smart probes and probing the system smartly.

<https://smartqa-academy/courses/smartqa-using-hybi>



**doSmartQA - AI based Smart Probing Assistant** to interrogate, hypothesise issues, design & evaluate user story or a set of stories in a sprint rapidly. An assistant for smart session-based testing based on HyBIST.

Download personal edition from [here](#)



**SmartQA Musings** - A gentle flurry of interesting thoughts on smart assurance as a weekly webcast. A refreshing view of assurance to broaden & deepen thoughts/actions.

[www.stagsoftware.com/subscribe](http://www.stagsoftware.com/subscribe)




**SmartQA Biweekly** - Ignite your curiosity with fresh insights, thought-provoking ideas, and inspiring content delivered straight to your inbox every fortnight.

[www.stagsoftware.com/subscribe](http://www.stagsoftware.com/subscribe)


## A rich collection of original content on smart assurance

**SmartQA Wisdom** - Profound nuggets of wisdom to think deeply, do rapidly & smartly for Test Practitioners.

<p>SmartQA Wisdom</p> <p><b>ON SMART UNDERSTANDING</b></p>  <p>Thiruvengadam Ashok STAG Software</p>	<p>on Smart Understanding</p> <p>on Smart Assurance</p> <p>on Personal Growth</p> <p>on Mindset &amp; Habits</p> <p>on Problem Solving</p>
---	--

Download from [www.stagsoftware.com/smartqa-wisdom](http://www.stagsoftware.com/smartqa-wisdom)

**SmartQA eBooks** - for Sr Engg/QA managers, Sr Test Practitioner & Young Test Practitioners too.

<p><b>HyBIST at a glance</b></p> <p>do SmartQA -The HyBIST Approach</p> <p>High Performance QA</p> <p>50 Tips for SmartQA</p> <p>Communicate Clearly</p>	 <p><i>Design smart probes via hypothesis based core method. Probe smartly in short immersive sessions.</i></p> <p><b>HyBIST at a glance</b></p> <p>THIRUVENGADAM ASHOK</p>
--	---

Download from [www.stagsoftware.com/smartqa-ebooks](http://www.stagsoftware.com/smartqa-ebooks)