WHITE PAPER | AUGUST 2016

The Top Five Misconceptions About Model-Based Testing

Is model-based testing simply too hard, slowing projects down and preventing true agility? Or does it offer a way to make testing more systematic and reactive to change, while also fostering close collaboration between the business and IT?

Huw Price
Application Delivery

ca technologies

Table of Contents

Section 1: Introduction 3

Section 2: All testing relies on models anyway 3

Section 3: Misconception 1: Our system is too big/too complex to be modeled. 4

Section 4: Misconception #2: We don't have time to create a model at the start of every iteration or project. 7

Section 5: Misconception #3: Maintaining a complete model cannot keep up with change. 8

Section 6: Misconception #4: Formal modeling is too hard for anyone but hardcore techies. 10

Section 7: Misconception #5: It's not agile/can't fit within an agile framework. 11

Section 8: About the Author 12

Section 1

Introduction

Model-based testing is not new, yet its application in testing remains limited. This is especially true since the advent of agile, and it is even rarer among teams who either adopt pure agile or take a hybrid approach.

The CA Agile Requirements Designer team has long advocated model-based testing. Based on the questions, challenges and outright rejections we've encountered to this approach, we've identified five of the most common misconceptions:

1. Our system is too big/too complex to be modeled.
2. We don't have time to create a model at the start of every iteration or project.
3. Maintaining a complete model cannot keep up with change.
4. Formal modeling is too hard for anyone but hardcore techies.
5. It's not agile/can't fit within an agile framework.

In addressing these concerns, flowchart modeling is used to represent some of the benefits offered by model-based testing.

Section 2

All testing relies on models anyway

Often, the challenge is semantic and stems in part from the bad press that modeling received when it didn't deliver on its promises in the past. It's worth noting here that, as Paul Gerrard argues,[1] all testing involves the modeling of a system, and the difference is therefore whether the modeling is implicit or explicit.

Take the example of behavior-driven development (BDD), which is currently more widely accepted than model-based testing, and yet relies at least implicitly on modeling. BDD starts with eliciting knowledge from core stakeholders and storing it in behavior-driven requirements, for example by using the Gherkin specification language.

The requirements are not then necessarily stored as a formal model, but when it comes to testing, testers will form models implicitly. When performing negative testing, for instance, they might consider what happens when a trigger in a Gherkin specification does not occur but the setup clauses have. If all test case design depends implicitly on models in this way, the question becomes whether there are advantages to making models explicit.

Section 3

Misconception 1: Our system is too big/too complex to be modeled.

If a system can be broken down into its cause-and-effect logic, it can be modelled. A flowchart, for instance, can break a system down into a series of "if this, then that" statements, linked to a hierarchy of processes. Any system is, in practice, constituted by such logical steps.[2] The question of whether model-based testing should be adopted is therefore one of whether there are practical benefits of doing so when faced with large or complex systems—as opposed to using alternative design methods. We outline some of these benefits below.

Componentization and abstraction

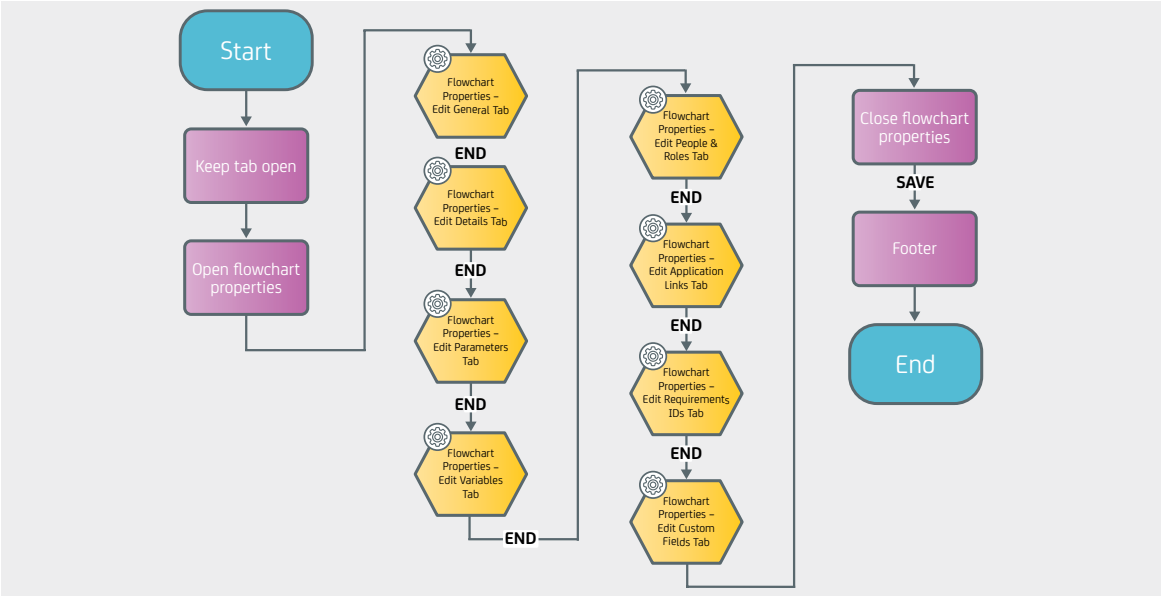
If a system really is that big or complex, modeling it can make the logic more manageable. No one person has to model the whole system; reducing it down to its logical components enables abstraction and componentization. Domain experts can instead set out only the parts of the system where they have expertise.

The individual components can then be incorporated under a hierarchy of higher-level processes, for example linking together subprocesses under a master flowchart. Individual stakeholders can drill down into as much granularity as they need to fulfill their roles. Higher-level flows might be used by testers and business analysts (BAs) to gain a broad understanding of a system and how its components integrate, while an individual tester might look in more detail at an individual component to gain the understanding they need to test it.

While only domain experts specify the functionality that they fully understand, another advantage of this approach is reusability. Once a component has been modeled, it can be shared and used by other testers in their models. As figure 1 shows, this can be as easy as stringing together subflows into a master flow. The repeated components and functionality found in a system can thereby be quickly assembled to create new tests. For example, a button or field which is used across user interfaces might be modeled and shared from a central repository. Similarly, in API testing, individual APIs might be modeled and tested before stringing together the subflows to form more complex types of tests like chain tests.

Figure A.

Reusable components, shown in yellow, have been linked together to create a master flow. Each yellow block is a lower-level process modeled as a flowchart.



Completeness

It's when you consider the alternatives to starting from a complete model that the practical benefits of formal modeling become clear. Agile methodologies and continuous delivery have changed the way requirements are gathered. Now, testers often face a constant barrage of change requests and user stories, rather than the written specification that stood at the start of a waterfall project. This reflects an effort to keep up with the rate of changing user needs, but brings with it a new challenge: Testers have to piece together the multitude of unconnected stories into a coherent system, including dependencies across components.

Incompleteness is a common consequence of designing a system in this ad hoc manner, using linear, unconnected stories. Even a relatively simple system is likely to have thousands of paths through its logic, so that connecting the dots between the disparate stories is near impossible unless performed in a systematic manner. Tests derived from such requirements typically only cover a fraction of the intended functionality and negative testing is usually especially neglected. In fact, our audits of test cases have revealed that just 10–20 percent functional coverage is the norm.

Using formal modeling to eliminate incompleteness

The process of systematically modeling existing requirements and user stories can help tackle incompleteness. Connecting linear stories into a coherent system forces the designer to think in terms of overlapping functional logic, considering the possible inputs and outputs as well as what must happen when a trigger is absent.

If a logical path hasn't been set out in the requirements, this will be abundantly clear. For example, a block in a flowchart might only have a single output. Because a flowchart model is mathematically precise, you can also apply automated predicate and completeness validation to identify any outputs missed during the modeling process.

It's far easier to spot any remaining incompleteness in a visual model, compared to a large written document or a series of unconnected diagrams. At one organization we worked with, a tester was mapping an ETL routine to a flowchart when a subject matter expert (SME) walked past their desk. At a glance, the SME identified a gap in the functional logic. With CA Agile Requirements Designer, user stories can also be derived directly from the model, and can be visually verified with a user or SME. This quick technique offers a reliable way to identify if any desired functionality has been omitted, validating each logical step in a selection of user stories.

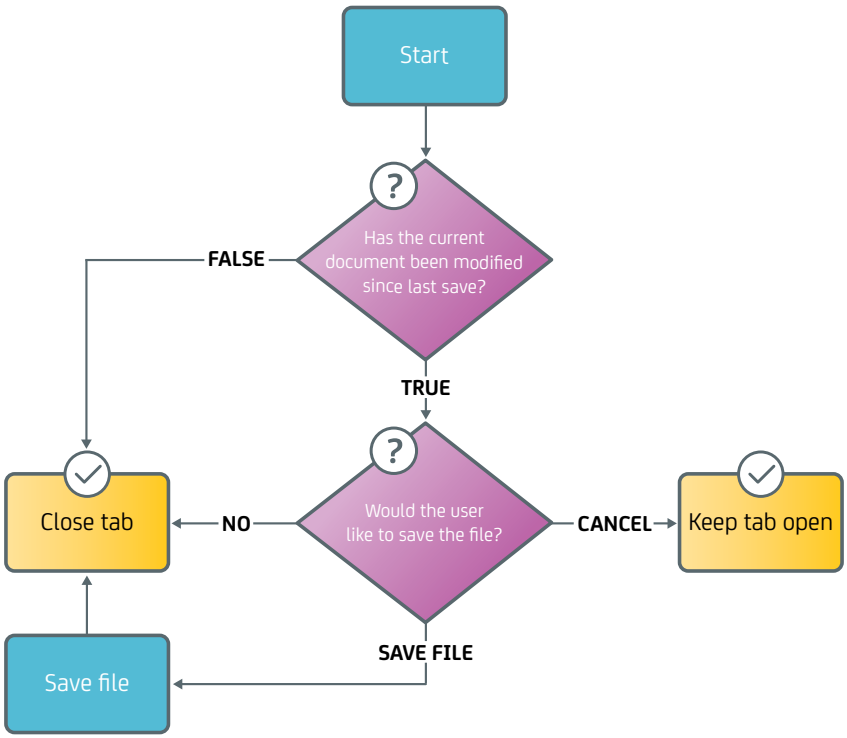
Using formal modeling to eliminate ambiguity

Along with incompleteness, many design methods rely on natural language—from traditional, written specifications to the language housed in BPM diagrams and user stories. Such natural language can lead to ambiguity and multiple interpretations, and is far removed from the concise, logical steps of the system that must be reflected in the tests. Misinterpretation of such written requirements frequently creates defects.

By contrast, a formal flowchart model visually breaks down the wall of words into a system's core, logical steps, which closely mirror a system's functionality. Therefore, it helps to avoid the defects created by ambiguity, which one study from 2009 estimates account for 56 percent of all total defects.[3] Similarly, a 2001 case study conducted by the IT University, Copenhagen, found that requirements defects accounted for 59 percent of defects at manufacturer B&K,[4] and a further study by the Hyderabad Business School from 2012 estimated that design activities can introduce as many as 50–65 percent of defects.[5]

Figure B.

A flowchart breaks a system down into its cause-and-effect logic. In this high-level flow, the functionality behind closing a tab in CA Agile Requirements Designer has been modelled, setting out when the system should prompt a user to first save their work.



It's worth noting that these studies span back to 2001, coinciding with the Agile Manifesto. In spite of the developments in how software design and testing is approached and viewed, the figures are consistent on the frequency and cost of requirements defects. They point to room for improvement within software design, with the potential to substantially reduce defect creation and remediation time and costs.

This was the case at B&K, where requirement defect prevention methods reduced usability problems by 70 percent and implementation defects by 20 percent. The project was also the first to ever be delivered on time. In other words, formally modeling requirements upfront offers manifest benefits, reflected in an improved user experience and on-time delivery, as well as a potential reduction in project cost by 64 percent, according to the research from the Hyderabad Business School.[6]

Section 4

Misconception #2: We don't have time to create a model at the start of every iteration or project.

Whatever time is spent formally modeling the requirements, it will almost always be outweighed by the time saved on manual test case design. Model-based testing shifts left the bulk of testing effort into the design phase so that creating test cases, data, automated tests and expected results can be automated.

Manual test design wastes time and compromises quality

Static, written requirements and flat diagrams don't accommodate automated test-case design. Instead, testers manually derive tests, storing them in spreadsheets or in a test management tool. This is a laborious, slow process, and one team we worked with spent 6 hours creating just 11 tests. As long as 70 percent of testing is manual,[7] it will rarely keep up with changing requirements and will almost always roll over to the next sprint.

Manually deriving test cases from static requirements is also unsystematic and error-prone, and can lead to a pileup of whichever tests occur in the minds of testers. As mentioned, even a relatively simple system will have more possible paths than even the most talented test team could think up. In audits of manually derived test cases, we've typically found that just 10–20 percent functional coverage is the norm. In fact, of 112 people surveyed by CA in July 2015, 42 percent cited a lack of test coverage creating defects and rework as a main software challenge.

Automating the test execution phase will not eliminate the time spent creating tests, and many automation frameworks rely heavily on scripting. Nor does it provide a way to test the potentially critical functionality which did not occur to the test teams when writing tests.

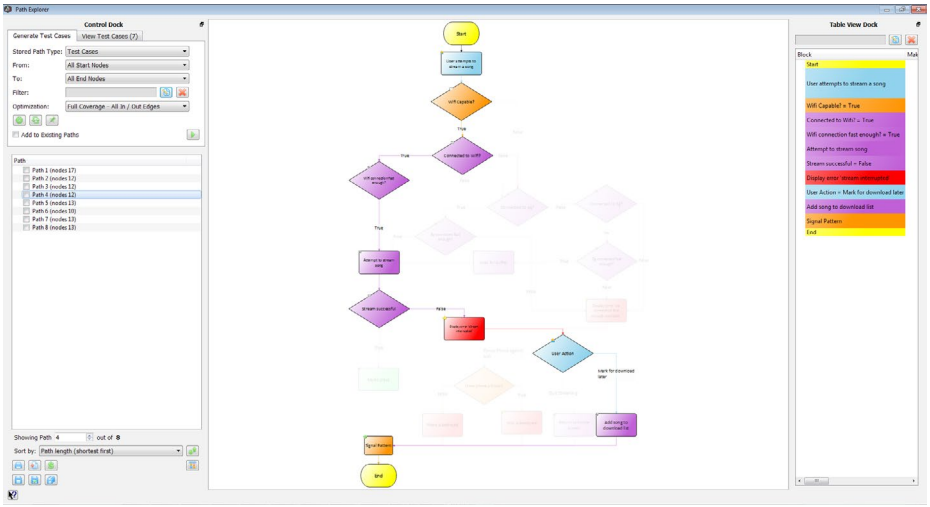
Automatically derive optimized tests

Model-based testing offers a way to automate the bulk of the testing effort, beyond just the execution, while also introducing the systematism needed for optimal test coverage. Because a flowchart is a mathematically precise model, underpinned by the functional logic of a system, automated graph homotopic analysis can be used to identify every possible path through it. These paths are equivalent to test cases and will reflect all the functionality set out in the design to ensure maximum functional test coverage.

Where it gets clever is with test optimization. When test cases are derived manually and linearly, the same logic will be repeated in multiple test cases—our audits have found that over-testing by a factor of four is the norm. With model-based testing, the test steps can be automatically combined in different orders to create the smallest set of paths needed for maximum coverage. Multiple optimization methods exist, and in one example, CA Agile Requirements Designer reduced the number of tests from 326 to 17, while improving coverage to 100 percent.

Figure C.

This model represents a simplified version of a mobile song streaming service. There are 71 possible paths through it, which are equivalent to functional tests and can be identified automatically using graph analysis. Using the all in/out edges coverage technique, the number of paths can be brought down to 8.



ca technologies

Automatically deriving tests in this way takes a fraction of the time demanded by manual test design, and further shortens execution. In one instance using CA Agile Requirements Designer, it took 90 minutes to create 108 test cases that provided 100 percent functional coverage—and this included time to create the flowchart.

Optimized tests created in CA Agile Requirements Designer can also be pushed out to automation engines, either using a framework like Critical Logic's TMX to convert the paths of the flow into scripts, or by overlaying code snippets into the individual nodes. These reusable snippets are then automatically compiled by the Path Explorer into the smallest set of tests needed for maximum coverage, complete with the right data.

Does modeling really take that long?

Another point worth making is that modeling existing requirements doesn't necessarily take that long. At Dutch insurance company a.s.r., a project manager approached a test team and gave them just two weeks to test a new system from scratch. Rather than attempt the impossible and manually write test cases, the test lead spent four hours modeling the requirements as a flowchart. Within a week, the team derived all the test cases needed for maximum coverage and synchronized them automatically with the existing HPE ALM framework. The 137 tests took two days to execute and the team completed the rigorous testing within the tight deadline.

You can further leverage existing requirements and test assets for an even faster design process when a modeling tool can integrate with an organization's range of tools. CA Agile Requirements Designer can import test cases or user stories from HPE ALM, CA Agile Central, VersionOne and Microsoft Team Foundation Server, in addition to tickets from JIRA and requirements stored in BPMN and XPDL-compliant tools. You can then import and consolidate the logic included in these disparate formats into a single model, eliminating incompleteness and ambiguity while enabling automated test design and execution.

Section 5

Misconception #3: Maintaining a complete model cannot keep up with change.

This misconception ties in with the complete specification fallacy, which suggests that maintaining a complete specification makes it impossible to keep up with changing user needs. In our experience, the opposite is usually true.

Traditional testing techniques cannot keep up with change

When faced with a barrage of static and fragmented requirements and requests, there's no way to automatically identify the impact that a change will have across a system's components; testers often try to work this out manually. This is especially difficult with technical debt, which grows as composite systems become more complex and subject matter experts leave without leaving behind solid documentation. Without knowing how all the moving parts in a system relate, an apparently simple or innocuous change might then cause a system to fall down, while a relatively low-priority improvement might end up requiring months of work.

There's also the effort of maintaining the test assets to reflect the change. Testers often check and update existing test cases by hand, but this is highly tedious and rarely achieves the coverage needed to validate that a change has successfully rippled up and down a system. Sometimes, tests are simply piled up in an effort to retain coverage, but this leads to rampant over-testing while invalid tests create time-consuming test failures. On other occasions, teams "burn" all existing tests and start again, but as the number of tests needed to test a system grows, teams quickly find themselves unable to keep up with the sprint cycle.

Automated dependency analysis and test maintenance

To keep pace with changing requirements, previous testing assets must be reactive and reusable. They should be leveraged when a change is made, and no two stakeholders should have to repeat one another's efforts. Flowchart modelling makes this possible, which, though complete, can be active and enable dependency analysis and automated test maintenance.

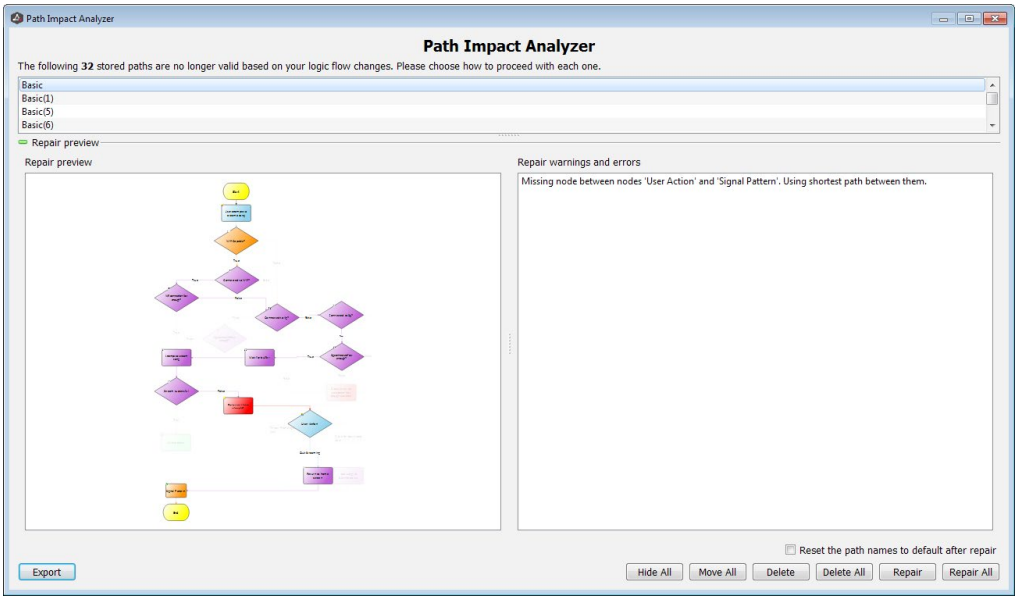
When a new piece of logic is added to an active flowchart, its impact on the paths within and across components can be identified automatically and mathematically. Automated dependency analysis can be used to identify what in a master flow will be impacted and how the change might reverberate in lower-level flows.

Using CA Agile Requirements Designer, you can perform this analysis prior to actually making a change to avoid unforeseen consequences. "Show impact" functionality also helps identify the way in which a change will affect the complexity of testing and developing a system. The relative value of a change can thereby be assessed before any unforeseen rework is created, in a way that's not possible with static design techniques.

The affected paths can be then updated automatically, using analysis similar to that used during path creation. The Path Impact Analyser feature within CA Agile Requirements Designer removes or repairs any broken or invalid test to avoid automated test failure and over-testing, and any new tests needed to validate the change will also automatically be created. This means that testers only execute as many tests as are necessary to validate a change.

Figure D.

The Path Impact Analyzer feature in CA Agile Requirements Designer has identified that some logic, which has been removed from the model, has rendered 32 paths invalid. By clicking "Repair All," the paths will automatically be updated, and any redundant tests removed.



This approach maximizes the value of work done, while also avoiding the bottlenecks created by manual maintenance—quickly providing an up-to-date regression pack of tests and data, and making rigorous testing possible within a sprint. Close collaboration is further fostered between technical and non-technical stakeholders who work from the same page. For instance, when a BA makes a change to a flowchart model, that change is automatically reflected in the test cases.

Section 6

Misconception #4: Formal modeling is too hard for anyone but hardcore techies.

This challenge seems to be a view held over from when formal modeling really was hard-core logic specification,[8] only suited to the most adept techies. The granularity offered by such modeling techniques is immense, as is their ability to build observability into the requirements and detect defects. Therefore, their use is imperative in industries like aerospace and defense where detecting defects during operations is mission-critical.

However, such methods equally make business stakeholders gawp[9] in a way that's at odds with the iterative development practices deployed by most organizations outside of these specialized industries. Fortunately, formality can be introduced to formats that are familiar to non-technical stakeholders, such as BAs.

BAs widely use flowcharting tools like VISIO and formats like BPMN, but flowcharting can also be mathematically precise, and with the right tooling, can include sophisticated logic like loops and non-linear constraints. A flowchart can thereby mirror the exact logic of a system under test; this precision enables testers to efficiently derive tests which reflect the requirements, as set out above.

It's true that a non-technical stakeholder is unlikely to be able to create a model with the granularity needed to derive executable test cases from it, along with test data and expected results. However, the value of a flowchart model is in the collaboration, where technical and non-technical stakeholders can work from the same page with different levels of abstraction, depending on the needs of their role.

A BA can work with higher-level flows, in a format or tool familiar to them, with the designs imported to CA Agile Requirements Designer. BPMs, written documents and requirements can all be imported. If the BA makes a change to the original requirement, difference analysis can be used to identify the change, reflecting it in the formal flowchart and updating the test assets accordingly. To avoid repeating the effort of translating requirements into tests, the tester leverages the design work performed by the BA.

In the "Bloor Market Report: Test Case Generation,"[10] Philip Howard assesses the strengths and weaknesses of numerous test case generation techniques, favoring flowchart modeling because it offers the benefits of formal approaches while being accessible to non-technical stakeholders. He describes how flowcharting enables the optimization of test cases offered by pairwise approaches, but also introduces cause-and-effect logic and traceability between requirements and test cases. Optimized tests can therefore be derived directly from the flowchart, and can be maintained automatically when it changes. Howard concludes that flowchart modeling is ultimately the easiest approach to implement in practice because it offers all the advantages of alternative approaches being "… much more (as opposed to slightly more) amenable to collaboration with business users."

For many organizations, the slight compromise in observing and detecting defects will be worth the collaboration between the business and IT. "Hardcore" specification techniques will simply not be suited to the agile development methodologies deployed by many organizations—but this is not to say that all techniques that incorporate formal modeling are too hard for non-technical stakeholders.

Section 7:

Misconception #5: It's not agile/can't fit within an agile framework.

This one incorporates many of the themes discussed already. It's worth stressing that model-based testing enables a shift left, where the effort of designing a flowchart automates many time-consuming tasks, such as test design, data provisioning and test maintenance. Rigorous testing is thereby made possible within a sprint because previous efforts can be leveraged, and a greater degree of testing is automated.

Flowchart modeling also brings business and technical initiatives into close alignment, and this is crucial within an agile context. With testers and designers working from a single point of reference, test assets derived from the requirements can be updated automatically to reflect any changes made to them. The otherwise linear stages of design, development and test are thereby collapsed into one, enabling a greater degree of parallelism and avoiding the defects and wasted time that occurs when a tester has to convert a design into test assets. A sprint really is a sprint, not merely a mini waterfall where testing is forced to the end and continually rolls over.

To learn more about model-based testing and see how it works in practice, watch the CA webinar, **"Which Came First—the Model or the Test?"** featuring testing guru Paul Gerrard.

Section 8:

About the Author

Huw joined CA in 2015 as vice president of application delivery, when specialist testing vendor Grid-Tools was acquired into the CA DevOps portfolio. During his 30-year career, Huw has gained a deep understanding of the challenges modern organizations face, and, with a thorough understanding of testing, how to solve them.

Huw has helped launch numerous innovative products that have recast the testing model. In 1988, he set up data archiving specialists, BitbyBit, and was soon joined by long-term partner, Paul Blundell. After BitbyBit was acquired, Huw and Paul co-founded data migration and application conversion firm Move2Open.

In 2004, they set up Grid-Tools Ltd, and Huw quickly began redefining how large organizations approach testing. He helped oversee the development of Datamaker (now CA Test Data Manager), pioneering a data-centric approach to testing, and later played a visionary role in the design and development of Agile Designer (now CA Agile Requirements Designer).

1 Paul Gerrard, "Models at Heart," CA Technologies, 2016, https://www.ca.com/us/collateral/white-papers/models-at-heart.register.html

2 Philip Howard, "Automated test case generation," Bloor Research, 2015, https://www.ca.com/us/register/forms/collateral/bloor-research-spotlight-paper-automated-test-case-generation.aspx

3 Bender RBT, "Requirements Based Testing Overview," Bender RBT, 2009, http://benderrbt.com/Bender-Requirements%20Based%20Testing%20Process%20Overview.pdf

4 Soren Lauesen and Otto Vintro, "Preventing Requirement Defects: An Experiment in Process Improvement," IT University of Copenhagen, Denmark, 2001, http://www.itu.dk/people/slauesen/Papers/PrevDefectsREJ.pdf

5 P Mohan, A Udaya Shankar and K JayaSriDevi, "Quality Flaws: Issues and Challenges in Software Development," Hyderabad Business School, GITAM University, 2012, http://www.iiste.org/Journals/index.php/CEIS/article/viewFile/3533/3581

6 Ibid

7 Philip Howard, "Automated test case generation," Bloor Research, 2015, https://www.ca.com/us/register/forms/collateral/bloor-research-spotlight-paper-automated-test-case-generation.aspx

8 Llyr Wyn Jones, "A Critique of Testing," CA Technologies, 2015, https://www.ca.com/us/collateral/white-papers/a-critique-of-testing.register.html

9 Philip Howard, "Bloor Market Report: Test Case Generation," Bloor Research, December 11, 2014, http://www.bloorresearch.com/research/market-report/test-case-generation/

10 Ibid

CS200-210233_0816