WHITE PAPER | MAY 2016

Using Model Based Testing to Drive Behavior-Driven Development

Huw Price
CA Technologies

Table of Contents

Section 1

Clarity and Collaboration as Driving Principles

As business and IT initiatives more closely align, the ability to communicate business requirements in a way that is directly transferrable to technical projects becomes imperative. As modern organizations rely increasingly on software that can provide value to its customers, IT teams are required to deliver fully tested software that delivers on changing business needs, faster, and for less.

By contrast, ambiguity in software requirements reduce the likelihood that IT teams will properly understand the software envisioned by end users and business analysts, in turn reducing the likelihood that software will provide value to the organization.

It is this frustration of developing software that fully delivers on a plausible interpretation of requirements, only to find out that it was not what was initially intended, that drove Dan North to advocate "Behavior-Driven Development". The constant "confusions and misunderstandings" he experienced led him to compare development to being led down a series of "blind alleys", where he would often be left feeling "If only someone had told me that!"[1]

Considering the challenges faced during typical IT projects, this frustration is evidently not unique among developers and testers. On average, only 4% of projects begin with clear requirements[2], and this in turn accounts for 56% of defects[3]. These defects are usually discovered late, during production, where it takes 50 times longer to fix them, compared to discovering them during the requirements stage[4]. The rework caused by ambiguity in requirements is therefore a reason why 60% of software projects fail or over-run, and poor requirements in fact cause 82% of the total effort spent fixing bugs[5], and cost a massive $250 billion in waste each year[6]. North was therefore justified when he "decided it must be possible to present Test-Driven Development (TDD) in a way that gets straight to the good stuff and avoids all the pitfalls" of miscommunication.

Behavior-Driven Development (BDD) seeks to foster collaboration between the business and IT, inspired by the notion of a 'ubiquitous language', taken over from Domain Driven Development. Given how IT teams are required to respond quickly to constantly changing business requirements, this is an agreeable concept: a common language, which can be understood by both core and incidental stakeholders (i.e. by those who focus on business objectives and application behavior, and those who implement these desired outcomes and behaviors), and which, by virtual of its semi-formality, can be readily translated into the logic of a system to be developed and tested.

BDD then becomes a matter of how "Behavior Driven Requirements" can be accurately formulated, so that they can be successfully implemented in development. This paper will consider just that. It will examine how these core principles of fostering collaboration and clarity between the business and IT can be best implemented during the development lifecycle.

Considering the process of requirements gathering commonly advocated, it will argue that introducing formal modeling can be synergistic to BDD, with the right tooling. The assumption that complete documentation (as opposed to "minimal documentation") necessarily means the inability to quickly respond to change will be challenged. Instead, it will be argued that formal modeling introduces the completeness of requirements, which, alongside unambiguity, is necessary to maximize the likelihood that software will deliver on the desired behavior of core stakeholders.

Section 2

Incompleteness

From the driving principles of BDD discussed, it is apparent that requirements must be both understandable by both the business and IT, and implementable as technical requirements, used by testers and developers. Capturing requirements through the interaction of those core stakeholders who will actually derive direct value from software, giving concrete examples or real-life scenarios of how they want an application to actually behave, appears on paper to be a sure-fire way to guarantee that requirements actually reflect the system envisioned by the business.

However, though this "outside in" development drives to eradicate ambiguity, the nature of the requirements and how they are derived raises questions for how implementable they are for testing and developing systems.

As argued by Llyr Wyn Jones in A Critique of Testing, the implementation of requirements can be considered a process of information transformations. Information is taken into one form, and it then output in another. For example, a developer transforms requirements into a piece of software when they code. The whole Software Development Lifecycle (SDLC) can be thought of like an information flow: the user works with a Business Analyst to produce requirements and use cases; the developer writes the code and then testers manually write test cases and test scripts. At each of these stages, information is passed on and 'translated' into a different form.

Building on the notion of requirements as active information, Llyr Wyn Jones defines ambiguity with reference to the concept of "uncertainty". This, he describes, leads to different information outputs, as the uncertainty regarding the meaning of the instructions lends itself to more than one way to implement the requirement. Likewise, incompleteness arises when necessary information about a system is left to assumption[7]. Both therefore lead to "uncertainty", and to the likelihood of misunderstanding between core stakeholders and incidental stakeholders.

In short, incompleteness, like ambiguity, reduces the likelihood that software will behave as the core stakeholders intended it. Incompleteness increases the likelihood of the sort of misunderstandings in development that Dan North sought to resolve, and so stands in stark contrast to the principles that led to the formulation of BDD.

Section 3

Incompleteness in Behavior-Driven Requirements

In the context of BDD, the information flow described by Jones occurs in the "feedback loop", whereby requirements are formulated, software is developed and tested, and then reports are passed back to core stakeholders to evaluate. In this context, the risk of uncertainty is that the "loop" will be repeated indefinitely, as software is constantly not delivered as intended, and as requirements constantly evolve.

By deriving requirements and the test scenarios on the basis of business objectives and desired behavior, testing emphasizes what should happen. In other words, it focuses primarily on happy paths. This can be seen in the natural languages proposed for writing scenarios.

If one considers a system's logic in terms of Boolean operators, then it can be wholly reduced to three functions: NOT, OR, and AND, with XOR a combination of all three. This in turn can be thought of in terms of IF ... THEN statements.

Gherkin, for example, places a greater focus on the "IF", "AND" and "THEN". It describes the initial conditions and set-up clauses for a scenario, as well as the "triggers" that are actually present (i.e., the "IF"), and the expected result of the scenario (the THEN). AND features in that each step can have multiple givens or triggers. The OR is apparently absent when gathering test scenarios in this format, and this can be seen more clearly in the alternative natural language used by Rspec, which might format scenarios in terms of a series of "When ... Then" statements[8]. In other words, such natural languages do not consider what happens when triggers are not present.

However, the decisions reflected by this "OR" are fundamental to a system's logic, and each represents a potentially distinct path, which the system will need to highlight as a part of testing. This is especially the case for negative paths, which should constitute around 80% of the total testing effort. Such unexpected results occur precisely when the triggers are not present, and so then "IF" is not fulfilled, while it is these unexpected results and outliers that are most likely to cause a system to collapse.

Some argue that BDD is a good way of identifying the paths which had been missed when designing the program, accounting for unexpected results if and when they occur. BDD is, after all an iterative process, and so shortening the "feedback loop" means that these negative paths can be dealt with sooner, in that testing is brought forward in the lifecycle. Some go so far as to suggest that organizations should "embrace uncertainty"[9].

As mentioned, however, leaving defects to be detected in testing is both costly and time-consuming, and in reality short iterations often become mini-Waterfalls, whereby testing rolls over or goes unperformed. There is also the more serious issue of the observability of defects. The expected result (Then) of a scenario is intended to be a verifiable condition, and testing is supposed to ascertain both that the expected result occurred, and that it was caused by the correct triggers, and arose from the correct set-up clause. But, when testing on an ad hoc basis, there is no way to know with confidence that results arose for the reason they are supposed to, and not because two or more defects cancel each other out.

Section 4

Model Based Testing and BDD

In order to move towards completeness, and increase the likelihood that software will deliver as intended, BDD needs to introduce the notion of "OR", in addition to "AND". The discretely presented scenarios of BDD and languages such as Gherkin then, should be connected up, in order to reflect the logic of a system. Doing so, testing in BDD will also consider what happens if the triggers do not fire when they should, taking a system down a negative path.

When writing and executing tests, testers typically do form models, albeit implicitly. When negative testing, for example, they might consider what happens if a trigger does not occur, but the set-up clauses have. Again, in BDD, if two test scenarios share a common step, they are implicitly connected by an "OR". They might, for example, share one trigger but not another, constituting a decision in the system's logic.

However, when this modeling is done in an ad hoc manner, testing is only likely to cover those scenarios that occur in the minds of testers and developers. This is the case in BDD when user stories are presented as discrete, linear units, which do not reflect how they relate to one another within the logic of a system. The functional coverage of testing will therefore remain low, typically 10-20%, as even a simple system is likely to have thousands of possible combinations of inputs and outputs—more than any one person is capable of holding in their head, and joining together accurately.

If BDD requires that testers model anyway, then there is no reason why this could not be done in a systematic manner, providing unambiguous and, crucially, complete requirements documentation. Such systematism is necessary if testing is to cover all possible paths through a system, including negative paths and unexpected results.

Section 5

Flowchart Modeling as a Part of BDD

Below is a proposal for how modeling can be incorporated into development, in a manner synergistic to the discussed principles of BDD.

1. Specification by Example. It is worth noting that requirements can still be driven by behavior, and derived from stakeholder interaction about how they would like a system to appear. Either the requirements formulated by the business are reverse-engineered as a flowchart, or the core stakeholders themselves build a flow. The latter is made possible because a flowchart serves as a ubiquitous language. Philip Howard of Bloor Research has described a flowchart model as something that can be underpinned by all the functional logic about a system needed by testers and developers alike, without leaving the business "gawping"[10].

   If developers and testers are using existing BDD requirements, the process is a matter of connecting up the overlapping steps of the scenarios, which then constitute decisions (OR's) in the system's logic. Doing this forces modelers to think in terms of a system's logic, enforcing completeness, while CA Agile Requirements Designer (formerly Grid Tools Agile Designer) will identify any broken paths, and provide path hints for any potential scenarios that have been missed. Missing or negative paths are therefore identified before the feedback loop is complete, reducing the number of iterations required.

2. Modeling Scenarios. User stories are, in reality, too high level for testing and development. Instead, the scenarios to be developed and tested are modeled in the flowchart, whereby the scenarios' steps form the process and decision blocks of a flowchart, and the scenarios become the paths through a system's logic. If all scenarios are modeled, every user story is likewise covered, as the scenarios are derivations of user stories.

   All the aspects of a narrative are readily transferrable to a flowchart. For example, a process block might specify who the stakeholder acting is, describing how "Customer attempts to withdraw money from ATM". The paths through a flowchart might then reflect a Unit Test in BDD, specifying a role, a desired feature, and a benefit or expected result.

   Because of combinatorial methods, there will not be a unique, distinct path from start to finish for each scenario. Instead, components of the functional logic of multiple scenarios or features will be combined in certain portions of the flowchart. This reflects an important difference between deriving tests from

BDD requirements gathered using natural languages such as Gherkin, and modeling a system's core functional logic: when modeling, a single test case can cover multiple scenarios, as these have been consolidated within a system as a whole, rather than being presented as discrete units. This consolidation of user stories as a complete system is a key step in moving toward reducing uncertainty.

Multiple scenarios, for example, might be consolidated up to a certain "Given" once a system has been modeled as a single flowchart. This "Given" could be a subflow, or a portion of multiple paths, which are then delineated by a decision block.

Similarly, several "Whens" might feature in any given path, in the form of decision blocks or process blocks. Multiple scenarios are thereby consolidated, in that if two or more scenarios share triggers, they can pass through the same blocks. Each decision on the way can then lead to a new path, or set of paths, with this continuing until a system is completely modeled, including all negative paths. For example, if two scenarios share all but one trigger, they could follow the same path until the very last decision block, where they diverge.

3. The feedback loop: verifying the model. If a system has been modeled by users and business analysts, then verifying the model is not required, and BDD can move onto to the next step. In this instance, the feedback delay will be very short indeed, as will be described.

   If a system has been modeled by testers and developers, verifying the model is similarly quick, and easy. In CA Agile Requirements Designer, it might be done with use cases. Each path through the flowchart will represent a distinct use case, which is equivalent to a test case or scenario once a system has been modeled. Verification is a matter of going through a set of use cases—presented in both the common vocabulary of plain text, and as a flowchart—confirming that the system is designed as intended.

4. The feedback loop: validating the model. Because the flowchart is underpinned by the functional logic of a system, test cases can be derived automatically from it. As mentioned, each possible path through a system constitutes a test case, and CA Agile Requirements Designer is capable both of automatically identifying possible paths, and calculating accurate metrics for the functional coverage they provide.

   What's more, invalid or redundant tests can be removed, and possible test cases de-duplicated, providing the smallest set of test cases required to provide maximum functional coverage. As noted, the logic of multiple scenarios can be consolidated, and so a set of scenarios might be tested using a smaller set of test cases—for example, 15 possible scenarios might be tested using 3 test cases. Test Optimization offers multiple algorithms to identify which paths provide maximum functional coverage. In one instance, for example, CA Agile Requirements Designer brought the number of possible tests down from 326 to just 17 with 100% coverage.

   Once these paths have been stored, they can be exported and executed. This might be done manually, pushing the test cases, linked to test data and expected results, to a test management tool such as HP ALM/QC. Alternatively, the paths can be exported as automated test scripts, being executed in numerous automation engines. In either instance, the "Living Documentation" and reports are produced by the test management tool used.

This automated process might appear similar to that of Cucumber, except the feedback delay is likely to be far shorter, while fewer feedback loops will be required. Firstly, automatically generating test cases removes the need for manual test case definition—in the case of Cucumber, it removes the need to manually convert Gherkin to step definitions in Ruby. In one instance, for example, it took CA Agile Requirements Designer 90 minutes to create 108 test cases, which provided 100% functional coverage. This included the time taken to design the flowchart; in practice, the time savings will be greater, in that the flowchart can be easily tweaked and re-used, as will be discussed.

Further, because developers have been provided with complete, verified, unambiguous requirements, they are more likely to deliver software as intended first time. As mentioned, ambiguity causes 56% of defects, and in this respect CA Agile Requirements Designer has reduced defect creation by up to 95%. Any defects that do make it past development to testing are also likely to be picked up first time, by virtue of the 100% functional coverage provided by the test cases. In addition to the time saved on re-work, test execution time is likely to be reduced. Duplicating tests has typically reduced testing by 30%, while test optimization can dramatically reduce the total number of tests needed to cover every scenario.

Section 6

Quickly and Easily Respond to Change

An objection might be put forward that flowchart modeling is going to be time consuming within a BDD environment, and the time could have been better spent running and re-running iterations of tests.

Firstly, it should be noted that this criticism only really stands if testers and developers are having to engineer flowcharts themselves: if business analysts and end-users favored flowcharting instead of using languages such as Gherkin, it would not exist. But, even if technical teams do have to reverse-engineer BDD requirements, the amount of time spent making a flowchart is generally small. Once the flowchart has been made, it can readily be re-used, even when the requirements change. This runs contrary to the assumption that complete documentation necessarily means the inability to quickly respond to change.

In CA Agile Requirements Designer, making a change to a flowchart is quick and easy; as users can simple add a new piece of functional logic to the flowchart. Any broken test cases are then automatically identified and repaired, while duplicate or redundant tests are removed. CA Agile Requirements Designer will then generate any new test cases required to provide maximum functional coverage.

This maximizes the value of the initial work-done—building a flowchart—and removes the time wasted manually checking every test cases by hand when a change is made. At one organization, it took two minutes to tweak an existing flowchart after a change request was made. CA Agile Requirements Designer automatically identified and repaired the three test cases that had been affected, leaving 64 untouched. As this example shows, flowchart modeling can further mean that only the aspects of a system which have been affected by change will be re-tested, helping to improve the efficiency of the feedback loop.

In sum, the time taken building a flowchart is most likely to be outweighed by the time saved on: rework and debugging caused by incompleteness and ambiguity; building test cases; executing test cases; implementing change requests. Complete documentation does not, therefore, stand in contrast to the principles of Behavior-Driven Development, but can better their implementation.

Section 7

Summary

Flowchart modeling provides a technique to introduce complete documentation into BDD, without compromising its core principles. The flowchart can be derived directly from stakeholder interactions, and is understandable to both the business and IT—it provides a ubiquitous language, which can be overlaid with all the functional language about a system needed by test and development teams. Such a flowchart can also help reduce "feedback delay", as testing cycles are shortened, and manual effort of testing is reduced.

Formal modeling increases the likelihood that software will deliver more accurately on requirements first time, as it provides a set of Behavior Driven Requirements that are both unambiguous, and complete. This reduces the total number of feedback loops needed before software delivers on the business requirements at any given moment, so that organizations can focus instead on innovation, rather than on re-iterations and frustrating rework. This ability to continuously develop solutions in an effort to maximize their value to customers is further facilitated by the ability to quickly and easily respond to change.

Section 8

About the Author

With a career spanning nearly 30 years, Huw Price has been the lead technical architect for several US and European software companies and has provided high-level architectural design support to multinational banks, major utility vendors and health care providers. Voted "IT Director of the Year 2010 by QA Guild, Huw has spent years specialising in test automation tools and has launched numerous innovative products which have re-cast the testing model used in the software industry. He currently speaks at well-known events internationally and his work has been published in numerous magazines such as Professional Tester, CIO Magazine and other technical publications.

Huw's latest venture, Grid-Tools, was acquired by CA Technologies in June 2015. For nearly a decade, it had already been redefining how large organisations approach their testing strategy. With Huw's visionary approach and leadership, the company has introduced a strong data-centric approach to testing, launching new concepts conceived by Huw such as "Data Objects", "Data Inheritance" and "A Central Test Data Warehouse".

Connect with CA Technologies at ca.com

CA Technologies (NASDAQ: CA) creates software that fuels transformation for companies and enables them to seize the opportunities of the application economy. Software is at the heart of every business, in every industry. From planning to development to management and security, CA is working with companies worldwide to change the way we live, transact and communicate – across mobile, private and public cloud, distributed and mainframe environments. Learn more at **ca.com**.

1 Dan North, Introducing BDD (2006), retrieved 06/03/2015 from http://dannorth.net/introducing-bdd/.

2 Chaos Manifesto 2013 (Standish Group: 2013), retrieved 07/03/2015 from http://www.versionone.com/assets/img/files/CHAOSManifesto2013.pdf.

3 Requirements Based Testing Process Overview (Bender RBT: 2009), retrieved 05/03/2015 from http://benderrbt.com/Bender-Requirements%20Based%20Testing%20Process%20Overview.pdf.

4 Why testing should start early in software development life cycle? (Software Testing Class: 2012), retrieved 06/03/2015 from http://www.softwaretestingclass.com/why-testing-should-start-early-in-software-development-life-cycle/.

5 Bender RBT, Requirements Based Testing Process Overview.

6 Kathleen Barret, Business Analysis: The Evolution of a Profession (IIBA: 2013), retrieved 05/03/2015 from http://www.iiba.org/Careers/Careers/Business-Analysis-The-Evolution-of-a-Profession.aspx.

7 See Erik Kamsties, "Understanding Ambiguity in Requirements", cited in Engineering and Managing Software Requirements (Springer: 2005), P. 250.

8 For an example of this, see http://en.wikipedia.org/wiki/Behavior-driven_development#Story_versus_specification

9 Dan North, Embracing Uncertainty (Goto Con: 2013), retrieved 06/03/2015 from http://gotocon.com/dl/goto-chicago-2013/slides/DanNorth_EmbracingUncertainty.pdf

10 Test Case Generation, retrieved 06/03/2015 from http://www.agile-designer.com/wpcms/wp-content/uploads/2014/12/Bloor-Market-Report-Test-Case-Generation1.pdf

CS200-196736_0516