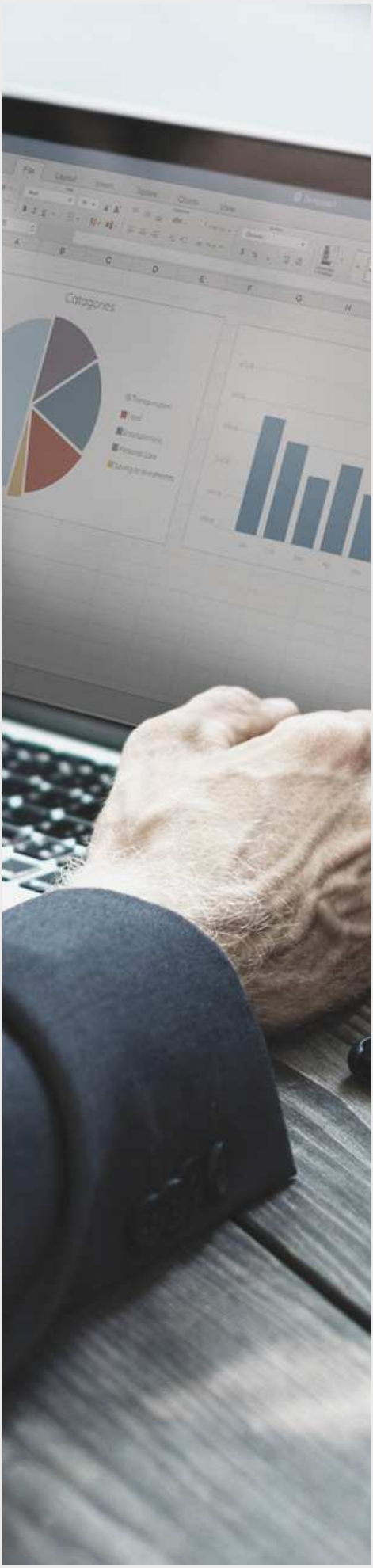
CONFORMIQ

Automate the Automation

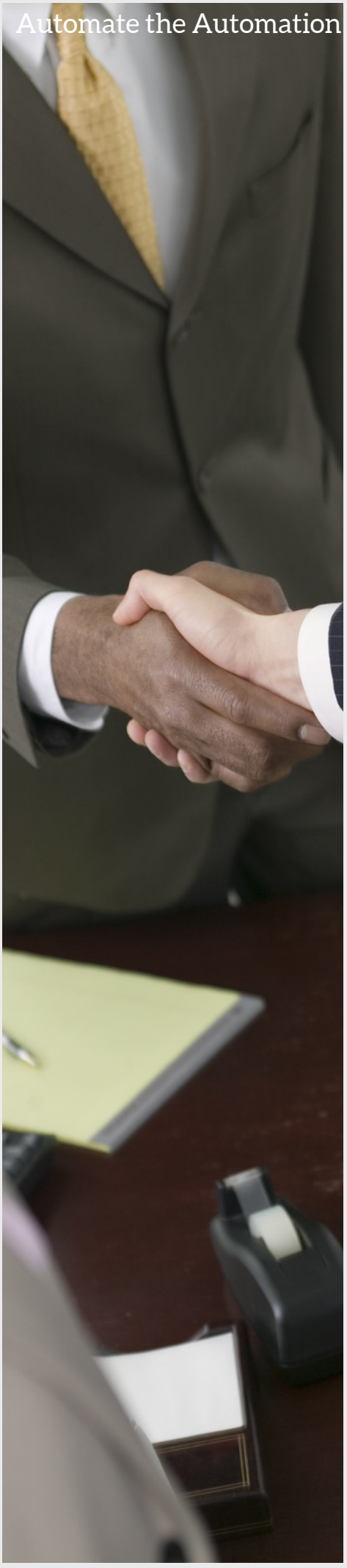


WHITEPAPER

AUTOMATED TEST DESIGN AS AN IMPROVEMENT TO TEST DRIVEN DEVELOPMENT (TDD) AND BEHAVIOR DRIVEN DEVELOPMENT (BDD) FOR AGILE TESTING

INTRODUCTION

There are multiple processes that have been and are being proposed for making functional test design faster than using traditional manual design techniques. It is primarily in conjunction with Agile development where these methods are getting the most attention. However, test design speed up is not the equivalent of improved test design productivity because there are other aspects necessary for overall improvement in the project's testing process. Even in Agile programs, the focus must be on improving the overall project's testing process, not just faster test design. Automated Test Design (ATD) solves this issue by automating the entire SDLC process in addition to automatically generating test designs and executable scripts at the speed of development. This paper is intended to provide more insight into these older developer test design methodologies, what they do and don't deliver, and then to compare and contrast them with the newer Automated Test Design process as implemented by Conformiq.



BACKGROUND

One of the processes promoted for speeding test design is Test-Driven Development (TDD). It is a core part of Extreme Programming (XP) and other "light weight" development practices and, though not a core part of Agile development, is a common partner to Agile.

As originally described by Kent Beck, TDD meant that before a developer could add a feature in the software, he/she first must write a failure test case. Their next objective was to write the minimal code that would pass that test case. Once the test passes, they refactor the code making sure that the test still passes. More broadly, TDD is used to describe any process where tests for a feature are written before the feature.

In practice, TDD has been considered too unstructured and so Behavioral Driven Development (BDD), as an enhancement over TDD, is currently getting significant interest as being the "next" great test improvement process for Agile development projects. Both these methods were developed prior to automated testing tools and thus don't account for improved efficiency from advancing technology. Yet their greatest limitation is their inability to deliver a complete testing process.

Behavior Driven Development (BDD) is a software development process that aims to combine the techniques and principles of Test Driven Development and Object Oriented Design by leveraging ideas from domain specific designs. It has excellent philosophical goals and ambitions, as BDD fundamentally aims to engage all stakeholders in the software development process by enabling non-technical stakeholders, such as business analysts, system engineers, and customers, to contribute and collaborate in the process by writing user stories.

In principle, BDD is founded on the use of a simple and informal notation, which is very close to common language and based on the main concepts of features and scenarios. Scenarios detail the "desired behavior" for each feature, which are essentially acceptance tests in the form of user stories. Probably the most widely known and used notation is Gherkin, which is used by tools like Cucumber, FitNesse, and JBehave.

Here is a Gherkin example:
Feature: The online shop keeps track of goods in a shopping basket.
Scenario: Put an item into an empty shopping basket Given the shopping basket is empty When user adds one item to the shopping basket Then the shopping basket should contain one item.

It is easy to see what is happening here. But if you take another look at this example, Gherkin is really nothing more than a requirement and an informal test description. This informality is both a strength and a weakness when it comes to test automation. Automation frameworks revolving around Gherkin can only generate simple code stubs and they still require a significant amount of implementation by software developers in order to get these codes ready for execution. Each Gherkin scenario clause is "just text," or text created with the preferred wording by the Gherkin author. Since automation codes need to be written manually and mapped to that description, dealing with change management is a significant issue. Scenarios make it very difficult to assess the quality and completeness of your software testing.

For example, how much have you actually tested your application? The use of Gherkin does not guarantee systematic coverage of functionality. Have you really tested all the possible data combinations? Which data combinations actually make sense? Do you have scenarios that fully cover all decision points within your functionality to be tested? Have you considered boundary values?

CHALLENGES

The use of TDD and BDD testing has proven to work in an Agile process, but the results show that with these methods, improved speed comes at the cost of loss in quality and knowledge, especially the understanding of test coverage. Other issues include:

1. Although TDD was created to match the need of a software development process with short development cycles, the constant time to market pressure made it hard to maintain and constantly update the (regression) test suites.

2. Another concern was that the test cases written by developers were created to cover their own code. They did not fully cover the system operation, thus the operation of multiple code parts written by different developers all running together was not taken care of in any of the tests. This meant that additional system (E2E) test cases needed be written later in the process causing the system level defects to be found much later.

3. One other issue was that because the developers wrote their own tests, this took
time away from their writing code and reduced their design efficiency.

4. And then the process of having the developer test his/her own code goes fully against the tenants of the IV&V process. Systemic defects can slip through the developer's "blind spots".
The TDD/BDD testing process is shown below.

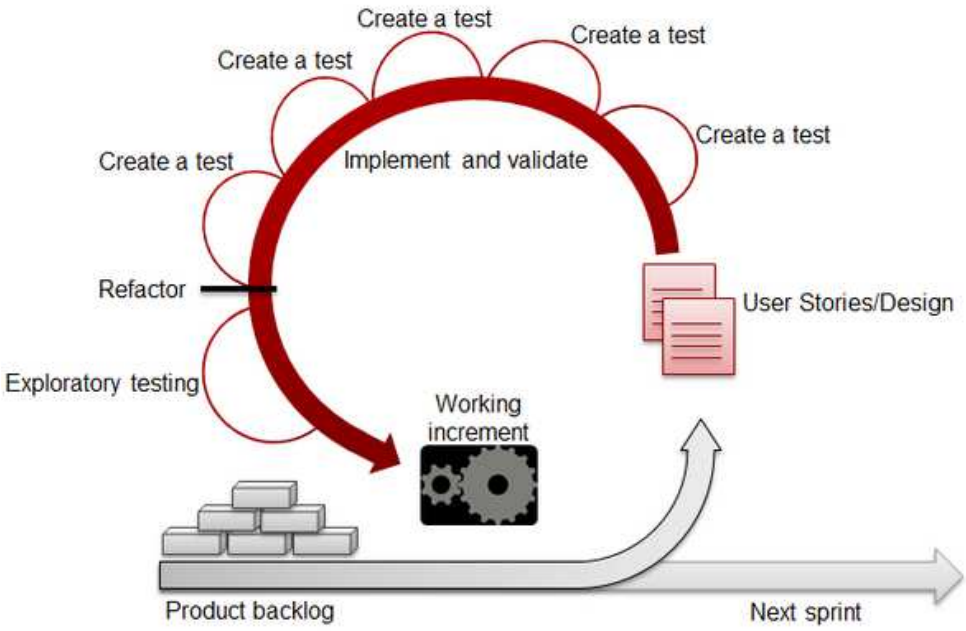


FIGURE 1. The TDD and BDD testing process

A MORE ADVANCED METHOD IS NEEDED

This is where Automated Test Design (ATD) based on Model Based Testing methods (MBT) and Conformiq come in. MBT enables testers and software developers to complement the work done by business analysts, system engineers, and customers by generating the tests they created as both Gherkin scenarios and test automation scripts of the entire test logic as code that includes test data. Tests are automatically derived and generated by the ATD tools, which provide systematic and repeatable coverage of the functionality to be tested. As requirements change, the model is quickly changed to match, and all generated scenarios and test automation codes are automatically updated to eliminate the issue of maintenance. Requirements can be directly downloaded from requirement management tools linked to the MBT models and requirement traceability automatically established.

Conformiq Automated Test Design is an approach to model-based black-box testing that starts with simple, high-level formal models of the system under test (SUT) that is being designed, and then automatically generates test cases. The model of the system can be continuously modified in parallel with development of the system itself. In TDD, you write a test case for a feature before you write the feature. When ATD is applied, you augment your SUT model to express your feature, then regenerate the tests (which will include one or more tests relating to your new feature), before you write the feature.

In on our experience with industry use cases of ATD, we have found two things: first, the productivity improvement of actual test case generation with ATD versus manual creation of tests is significant, on the level of an order of magnitude. Second, this productivity improvement is even higher in the context of incremental development: with ATD, the tool will automatically regenerate the full test suite when the model is changed, including determining which prior test cases are no longer applicable. Plus, ATD automatically generates the test oracle, i.e., the expected correct test execution result.

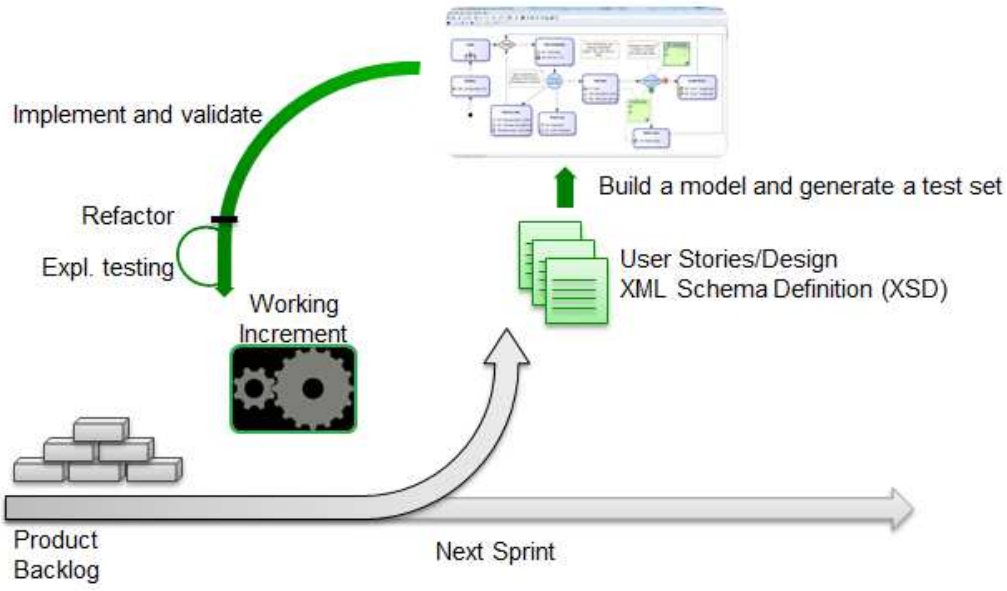


FIGURE 2: The Conformiq ATD testing method process

WHAT ARE THE TDD/BDD ISSUES THAT ATD SOLVES?

These experiences would argue that ATD is particularly well suited for Agile and as an improvement to test driven development methods. In addition, ATD can help in improving some of inherent issues of TDD and BDD that adherents have raised:

- Not well understood requirements. Here the argument is that it is inefficient to apply TDD as the requirements are not well understood early in the process. It is true that the requirements typically contain ambiguities, omissions and contradictions. One benefit of ATD is that just the act of modeling the system behavior often improves the quality of the requirements. That means a lot of defects can already be spotted in the model of the specifications and requirements before even writing a single line of code. As one creates a model of the system behavior, one often raises a lot of questions regarding the requirements, so already the modeling process can expose a lot of issues with the requirements. This should not come as much of a surprise. After all, system modeling involves the development of a small high-level prototype of the real system and it has been long known that prototyping is a good and efficient way of finding inconsistencies in the requirements.

- Varying requirements. This is especially important within Agile development projects where the requirements are updated during the project. In ATD, a simple formal model of the SUT will explicitly embody the requirements and then the refactoring of requirements is dramatically simpler to do than the equivalent effort of refactoring a set of manual tests. With ATD, the effort is linear with the number of requirements that change, whereas in a manual process it's proportional to the product of the requirements that have changed and test cases (since all test cases need to be checked for all requirements that have changed).

- TDD doesn't emphasize good tests. The argument here is that as the developers have not implemented the solution yet, the tests are not "good enough" and, for one, they do not explain the solution. With ATD the idea is that the model represents the actual, desired behavior of the system itself – not the test cases nor how it should be tested. ATD improves the quality of the test cases because the automated approach to test design lowers the risk of having incorrect, missed and redundant tests. An engineer can, for example, accidentally miss a test case that is dictated by the requirements, such as for an error handling case, a limit value of a data parameter, or an expiration of a rarely activated timer, but not so with the algorithmic approach.

- Unit tests are not system tests. TDD test cases written by developers cover their own code. They do not cover the system operation and the operation of multiple code parts written by different developers all running together. This means that additional system test cases must be written later with the TDD approach. System defects are found much later. With ATD, these system tests are automatically created as the model grows or models are combined into the full system.

- Over fitting tests to the code. A common concern is that if a developer first writes the tests, he may over fit the actual implementation to the tests. With ATD, the developer doesn't design the test cases, so there is no risk of fitting the implementation to the tests. However, a key point of model-based black-box testing is that the system is judged against an independent reference. Without this approach there is naturally a possibility that the developer reflects the same fault both into the test and then into the implementation code. This also highlights the importance of good software development and testing practices, such as model reviews as part of TDD and Agile.

- Tests are expensive to implement too early. Here the adherents say that the tests should be guided by code and expert knowledge of the implementation on where the problems might be. Therefore implementing tests too early is expensive as you do not have the details of the code available. This is a very common white-box view to the problem where we expect to have access to the implementation details for devising test cases. However, with model-based black-box testing we approach the problem from a different angle and we assume no implementation details of the system and we validate whether the given system conforms to its design and functional specification. The test cases that a model-based testing tool like Conformiq CreatorTM generates are black-box by nature which means that they depend on the model and the interfaces of the system under test, but not on the internal structure of the implementation. One does not require an understanding how the system has been architected internally in order to create a model, thus lowering the cost of test creation and allowing tests to be generated prior to incremental code drop.

- Not all developers know how to or want to test. Testing requires a different mindset from development and it may be true that some developers are poor in doing test design. This issue can be resolved by pairing the developers with people who know how to test. When ATD is applied, modeling can be accomplished by a non-developer (a SME, a modeler, etc.) who is an integral part of the development team. This means that developers do not need to also be testers and thus do not need to spend time writing test cases, a key benefit in an agile delivery process with short sprint times. This improves their efficiency but also, as no separate I&V function exists, having the same person write both testing assets and code sets up the potential for pathologic errors.

CONFORMIQ ATD SOFTWARE

Conformiq has developed an approach to model-based black-box testing that starts with simple, high-level formal models of the system under test (SUT) that is being created, and then automatically generates test cases without further user involvement or direction. The model of the system can then continuously be fleshed out in parallel with development of the system itself.

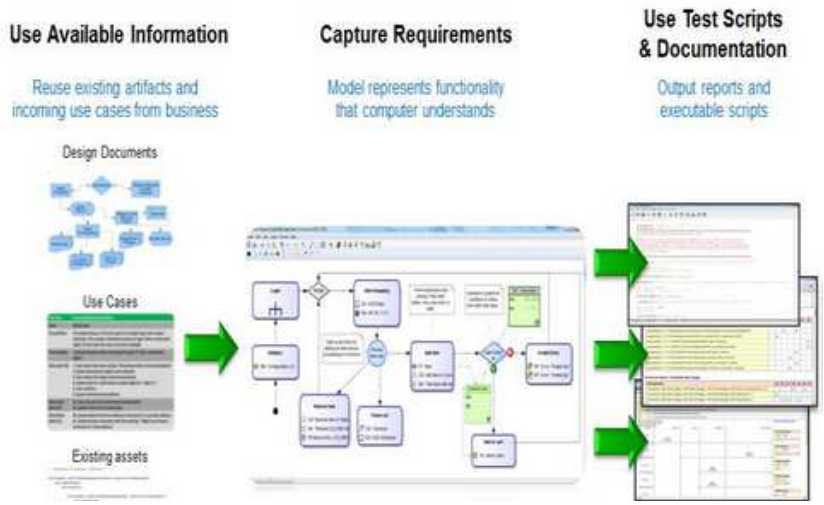


FIGURE 3: The Conformiq 3 step process

With Conformiq's Automated Test Design process the model is manually created from the requirements and test scripts are automatically generated for execution

The capabilities beyond test cases that are automatically generated by Conformiq are those needed to ensure that testing is done well and that the stakeholders have this knowledge. Further, because test cases are generated, they are consistent in how they are written, whereas manually created BDD textual test cases vary by author.

CAN THESE PROCESSES BE USED TOGETHER?

Instead of seeing BDD and MBT as competing approaches, we can view them as being complementary. It can be beneficial to integrate the two approaches in order to get the best of both worlds. Instead of requiring and relying on software developers to "connect the dots" through manual implementation of code stubs and maintenance of every clause in every scenario, the MBT model is automatically updated. The model is then processed by Conformiq's ATD engine which generates an optimal collection of tests that can be exported as Gherkin scenarios for business analysts, system engineers, and customers for model review. These can be executed automatically using frameworks such as JUnit, Cucumber, FitNesse, or also output in any other execution for QTP, Selenium, and other frameworks.

SUMMARY

The speed of agile development is a key to its appeal and broadening use. However, testing methods developed over a decade ago are insufficient for delivering quality at the speed of development. A new process of Automated Test Design eliminates the inherent problems with those previous developer-centric testing methods and instead introduces a method that delivers the needed quality and documentation for the project at the speed of development.