Automated Test Design as an Improvement to Test Driven Development

Test-driven development (TDD) 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 you add a feature in the software, you first write a failure test case; your next objective was to write the minimal code that would pass that test case. Once the test passes, you 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.

Conformiq Automated Test Design (ATD) 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 then continuously be fleshed out 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 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regenerate the full test suite when the model is changed, including determining which prior test cases are no longer applicable. The ATD process is shown in Figure 1.

![Figure 1: Testing is a three step process – model – generate test cases – output executable test scripts](image-url)
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 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 actually often so that the requirements contain ambiguities, omissions and contradictions. One benefit of ATD is that the mere 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 writes 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 requirement bugs.

- **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 refactoring of requirements is dramatically simpler to do than the equivalent effort in 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, for example 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, as 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 to the model and then to the implementation. 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 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 Designer™ 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. Even when ATD is not applied with TDD, this issue can be, for example, 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 a potential for pathologic errors.

The incremental nature of Agile is well matched with ATD. After we have made the updates to the model, a new test suite can be automatically generated. Test cases can be immediately executed if the harness is ready for automated progression testing, even if sprints are as short as two weeks. When regenerating the test suite, the tools establish an incremental traceability and can directly report which of the test cases were removed, which were added and which became redundant. The test during development concept embodied in TDD is then also a match to ATD. Both facilitate a “Shift Left” strategy of earlier testing to find and fix defects during development, not afterwards. ATD just also enables system testing to be accomplished during the development process while TDD does not.

Conformiq itself uses TDD with Conformiq Designer and Jenkins. The benefits and approach we have seen are reflected in the preceding paragraphs, making this article not a theoretical paper, but a rather documentation of our experiences.

**Conformiq Contacts**

Conformiq Inc. is headquartered in Saratoga, CA USA with offices in Europe and India partner operations. It develops and delivers advanced automated test design software.

Americas: +1.408.898.2140; Europe & Scandinavia: +358.10.286.6300; India, Asia, & Asia/Pacific: +1.408.898.2140 or email sales@conformiq.com, www.conformiq.com

Copyright © Conformiq Inc. 2013 All rights reserved.