Annales Univ. Sci. Budapest., Sect. Comp. 36 (2012) 75–98

IMPROVING EFFICIENCY OF AUTOMATED FUNCTIONAL TESTING IN AGILE PROJECTS

Gáspár Nagy (Budapest, Hungary)

Communicated by László Kozma

(Received November 30, 2011; revised January 16, 2012;
accepted February 10, 2012)

Abstract. *Test-Driven Development* (TDD) is probably the most important agile engineering practice. Since it was first described in detail in [1], this development practice has been adopted widely. This adoption has been also well supported with tools that provide a framework for defining and executing unit tests on the different development platforms. Test-Driven Development provides a guideline how to develop applications unit by unit, resulting in well-designed, well maintainable quality software. TDD focuses on units and it ensures that the atomic building blocks and their interactions are specified and implemented correctly. There is certainly a need for automating other tests as well in order to ensure a working integration environment, to validate performance or to ensure that the application finally behaves as it was specified by the customer. Usually for automating these non-unit tests, developers (mis-)use the unit test frameworks and the unit test execution tools. This paper investigates the potential problems caused by misusing unit test tools for automated functional tests in cases when these functional tests are defined through the development technique called *Acceptance Test Driven Development* (ATDD). The misuse of the unit testing tools may have direct impact on the testing efficiency and it may also "close the doors" for features specialized for automated functional tests. Some results of this investigation have been prototyped in a tool called SpecRun, which aims to provide better automated functional testing efficiency.

Key words and phrases: Test-Driven Development, Acceptance Test Driven Development, Behavior-Driven Development, Acceptance Criteria.

The Research is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1./B-09/1/KMR-2010-0003).

This paper was supported by TechTalk Software Support Handelsgesellschaft m.b.H.

1. Introduction

Behavior-Driven Development (BDD) [2] is a way of building software focusing on application behavior. *It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters* [3]. This is achieved by enabling a better communication between the customers and the development team and by using automated acceptance tests to describe the required functionality, using the technique called *Acceptance Test Driven Development* (ATDD) [4]. BDD is an outside-in methodology that uses *Test-Driven Development* (TDD) [1] to ensure a robust design for the application. In this paper – for the sake of simplicity – I will refer the automated functional tests that have been defined through the ATDD technique as "ATDD tests". Many of the statements or conclusions can be generalized to other automated functional tests or to integration tests. Only a few of them can be applied to automated performance tests though.

At TechTalk [5] we have developed the open-source tool *SpecFlow* [6] for the sake of a better support of the BDD process on the Microsoft .NET platform. SpecFlow is primarily a tool for automated acceptance testing, but – following the common practice – it uses unit test frameworks (NUnit [7], MsTest [8], etc.) for executing the ATDD tests.

In the last years I have been practicing the BDD/ATDD technique and have helped introduce SpecFlow to several projects. In almost all of these projects, as soon as the number of tests reached a certain limit, the problems with continuously executing these tests became more and more visible. By trying to find the root cause of these problems, we have found that in many cases these are somehow related to the misuse of the unit test tools.

During the problem analysis and the search for solution, I have tried to implement a holistic approach. For example, when improving the efficiency of the local test execution on the developer machine, I did not only consider the technical solutions achieve faster test execution by the machine, but also how the delay, caused by switching from the development environment to the test tool can be shortened; or how the number of tests executed in one round can be limited with a better test management process.

Several research studies have shown that testing efforts make up a considerable part (at least 50% [9, 10]) of the total software development costs. The long term maintanance costs can be as high as two-thirds of the total costs [11, 12]. Therefore, testing efficiency is a vivid topic in the reserach area both in the academic and in the industrial fields. These results provide sound results for areas like test-case generation [13], specification-based testing [14], test prioritization [15] or random testing [16]. The target of my research is

to improve testing efficency in the agile development process of medium-size (300 to 1000-person-day development effort) business applications that are not specified formally. Though the mentioned results can partly be used to improve the quality of these applications, they do not give any proper answer for improving the test-driven development process, where the human aspect plays an important role. This aspect is quite new and has not been thoroughly covered in literature.

In this paper, I am trying to address a small aspect of the improvements, the problem of test execution efficiency of automated functional tests in the test-driven development process. Though some parts of this improvement can be exactly measured, the majority of the results can only be seen from the content of the team members and stakeholders. My results are based on the feedback of several project teams at TechTalk. These developers and other stakeholders were heavily interested in improving the efficiency for the given conditions, hence their judgement is authentic.

The rest of this paper is organized as follows. After a short overview of the terminology (Section 2), TDD (Section 3) and ATDD (Section 4), Section 5 compares these two development processes. Section 6 categorizes the efficiency issues I have encountered in four main groups: execution time, feedback about the execution, execution history and test setup.

As we have learned more about these problems, TechTalk has decided to create a tool specialized for more efficient integration test execution, where the findings have been partly implemented. Section 7 provides a quick summary about SpecRun [17].

Section 8 lists possible improvements for these problems that are implemented or planned for SpecRun.

The paper finally provides a summary and an outlook for further improving testing efficiency (Section 9).

2. Terminology

The *Test-Driven Development* term is well established in the development community as well as in academic papers. This means that more or less they agree on the basic principles of TDD.

Unfortunately, the picture is not so clear when one enters the area of executable specification practices. The concept of driving the application development through automated functional tests has been established in the agile software engineering community under various names, like specification by ex-

ample [18, 19], story test driven development [20], executable acceptance tests [4, 21], or acceptance test driven development [4]. A good overview of the literature of this idea has been done by Shelly Park, Frank Maurer at the University of Calgary [20].

In this paper, I use the term *Acceptance Test Driven Development* (ATDD) to describe the technique of developing the application through automated acceptance test. I use the term *Specification by Examples* to denote the technique of describing acceptance criteria using illustrative examples, and finally the *Behavior-Driven Development* (BDD) that describes the holistic methodology of the application development that uses all these techniques in application development.

The terms *acceptance criteria* and *acceptance test* have similar meanings in the referenced literature. Neither of these terms is perfect, as both of them are easy to mix with *user acceptance tests* [22]. The term acceptance test has an additional disadvantage: the word "test" gives the wrong impression of referring to quality assurance and not to the requirements. In this paper (except for quotes), I use the term acceptance criteria to denote the specification element, and by acceptance test I mean the automated executable acceptance criteria.

3. Test-Driven Development

It is not the goal of this paper to describe TDD in details (as it is better described in detail in [1] and [4]). I would like to provide a short summary, though, focusing on the aspects that are the most relevant for a comparison with ATDD. This will cover the basic workflow recommended by TDD and a brief overview of the supporting techniques. Some detail aspects of TDD will be also briefly described in Section 5.

Test-Driven Development is based on a cyclic workflow that can be used to develop the small, atomic components (*units*) of the application. This workflow, which is often mentioned as red-green-refactor (Figure 1), is composed of three main steps.

Step 1: Write a Unit Test that fails. The failing unit test ensures that the unit test is able to indicate malfunctioning code. As the unit test execution environments display failing unit tests as red bars, this step is also referred as "red".

Step 2: Make the failing unit test pass. Implement the unit being tested (aka. *unit under test* - UUT) focusing on making the test pass in the simplest way possible. It is essential that the implementation goes only so far that the

test passes and not further. As with Step 1, this step is often referred as ”green” because of the usual indication of the unit test execution environments.

Step 3: Make the implementation ”right” with refactoring. The implementation provided in Step 2 might contain a code that is not ”right” (maintainable, clean, elegant) due to the goal of ”the simplest way”. In this step, this code has to be changed to shape it into a better form. Refactoring denotes here code changes that do not alter the behavior [23], so the test(s) that have been passing so far should still pass.

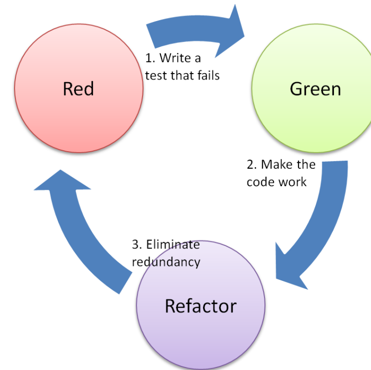

Figure 1. Red, green, refactor cycle

The unit tests used in the TDD process should follow a simple three-part structure, which is denoted with the acronym AAA, where the elements stand for the following (as described at [24]).

Arrange all necessary preconditions and inputs. This part of the unit test should set up all prerequisites that are necessary to execute the unit being tested.

Act on the object or method under test. This part is the actual execution of the method that should provide the expected behavior.

Assert that the expected results have occurred. In this part, different kinds of verification steps can take place.

Test-Driven Development focuses on small steps where the small parts of the code (units) are built up in a test-driven manner. The units are small and focus on solving one particular concern, furthermore, they should be isolated from other parts of the applications. This rule ensures that the unit tests driving the implementation of the unit can be kept within limits and they do not suffer from the *test case explosion* [25] problem. This rule is also very important in decreasing the dependencies between the parts of the code, which generally has a bad influence on the maintainability and the development process.

4. Acceptance Test Driven Development

Acceptance Test Driven Development (ATDD) is a technique to develop applications through automated acceptance tests. As a supporting technique, ATDD fits Behavior-Driven Development (BDD) [2].

The basic principles of ATDD are described by Koskela [4]. He defines acceptance tests as *specifications for the desired behavior and functionality of a system. They tell us, for a given user story, how the system handles certain conditions and inputs and with what kinds of outcomes.* He also enumerates the key properties of acceptance tests as:

1. *Owned by the customer*
2. *Written together with the customer, developer, and tester*
3. *About the what and not the how*
4. *Expressed in the language of the problem domain*
5. *Concise, precise, and unambiguous*

Finally, he describes the workflow of ATDD. Generally, ATDD also uses a cyclic workflow to implement functionality like in TDD. The workflow of ATDD consists of four steps:

1. Pick a piece of functionality to be implemented (e.g. user story [26] or acceptance criterion [27])
2. Write acceptance tests that fail
3. Automate the acceptance tests
4. Implement the application to make the acceptance tests pass

Figure 2 shows the ATDD workflow by Koskela, extended with the a refactoring step. In practice, the refactoring of the implemented functionality is as useful as for TDD.

ATDD is driven by the expected functionality of the application. Agile development processes focus on delivering business value in every step, so the expected functionality has to be exposed in a facade where the stakeholders can realize the business value. A new database table or an application layer very rarely has a measurable business value. In agile projects, the application functionality is usually defined on the basis of what can be observed on the external interfaces (e.g. the user interface) of the application. Because of this, it is obvious that the acceptance tests should also target these external interfaces.

The implementation of even a small aspect of the functionality (acceptance criteria) is usually too complex to fit into a simple unit (e.g. it exercises the different layers of the application). Usually a cooperation of several units is necessary. Therefore ATDD does not replace the concept of implementing the units in a test-driven manner, on the contrary, it embeds this process for developing the units [4]. As this collaboration of techniques is a key part of ATDD, the ATDD workflow (outlined by Koskela) is usually represented as a two-level nested cyclic workflow.

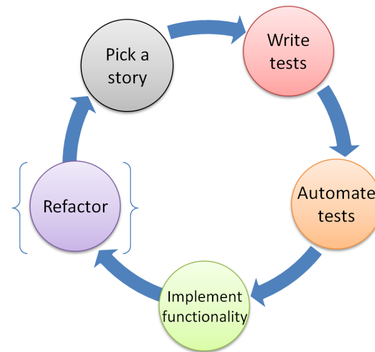

Figure 2. Extended ATDD workflow

5. Key differences in the application of TDD and ATDD

As I described before, the ATDD workflow has inherited a lot from TDD, so the similarities are conspicuous. At the same time, the deeper investigation of the methodologies shows also some differences. This section briefly enumerates through these differences in order to explain some efficiency problem in Section 6.

1. ***ATDD tests are integration tests***
   The most obvious difference is that acceptance tests ideally test the functionality end-to-end, integrated with all layers and dependencies.

2. ***The definition and the implementation of the acceptance criteria are accomplished in different phases***
   In ATDD, the acceptance criteria (that are the bases of the acceptance tests) are defined at a different (slightly earlier) stage of the development process (e.g. in the sprint planning), so the implementation of one acceptance criterion cannot influence the definition of the next criterion, like in TDD.

3. ***People who define acceptance criteria are usually different from the people who implement the application fulfilling these criteria***
   The specification is mainly done by the business, the developers can only influence them by giving feedback about the feasibility.

4. ***In ATDD an early feedback for the "happy path" is required***
   When implementing functionality, it is a good practice to focus on the most common scenario ("happy path" [28]) first. This is the best way to receive quick and valuable feedback from the business.

5. ***ATDD tests do not provide a complete specification of the application***
   In business applications, where ATDD is commonly used, the written specification is not complete and the part of the specification that is formalized into acceptance tests is even less so. To be able to implement the application based on these, everything that was not specified should be "common sense" behavior.

6. ***ATDD acceptance tests are black box tests, while TDD unit tests can be white box tests***
   As the acceptance tests are driven by the required functionality, they are more like black box tests.

7. ***Acceptance tests should be understood by the business and testers***
   The acceptance tests are about the functionality; in order to verify whether the formalized acceptance test really describes the intended application behavior, the business representatives should be able to read and understand the tests.

8. ***The implementation of an ATDD cycle might take several days and several developers***
   The implementation of even a small aspect of the functionality (acceptance criteria) is usually complex (e.g. it exercises the different layers of the application). Therefore, it can happen that it takes several days and several developers to complete.

9. ***The execution of the ATDD tests might take a long time***
   As mentioned before, the ATDD tests are integration tests and the execution time of the integration tests is usually much longer than that of a unit test. This is mainly because these tests have to initialize and use external systems (typically a database system).

10. ***The analysis of a failing ATDD test might be accomplished much later than the development***
    Since the execution of the tests takes a long time and the developers have probably started to work on another task in the meanwhile, the failing ATDD tests are not investigated and fixed promptly.

11. ***ATDD tests might be changed by non-developers***
    Though in most of the environments this is not common, in some cases

even the business analysts and testers change the acceptance tests. Usually these changes concern the expected result values or the adding of further test variants (input / expected output pairs) to an existing test.

These differences can be observed in almost every team using ATDD tests. From these differences it is visible that, although the base concept of TDD and ATDD is similar, there are also many differences. Using unit testing tools for ATDD tests is typical, but due to these differences, it can lead to testing efficiency issues. The following sections describe these problems and give ideas for solutions.

6. Efficiency problems of executing functional tests

As mentioned earlier, in the projects in which I participated the problems of the continuous test executions became more visible once the number of tests reached a certain limit. Of course, this limit cannot be exactly defined, but generally the problems become more visible when

1. the tests execution time on the continuous integration server exceeds 30 minutes
2. at least half of the test execution on the server fails due to a transient error
3. effort spent on analysis of test failures on the server becomes significant
4. test execution time of the work-in-progress tests on the development machine exceeds 10 minutes

By trying to find the root cause of these problems, we have found that in many cases these are somehow related to the misuse of the unit test tools. These tools are specialized for executing unit tests that are fast, isolated, stable and predictable. In the case of ATDD tests, these conditions are usually not fulfilled.

When planning for addressing these issues with a specialized tool, we have made a questionnaire to collect feedback about the functional test execution problems. The questionnaire was filled by a dozen of software development companies that use ATDD extensively. While the result is certainly not representative, it gives a good external overview about the problems. In the questionnaire we have listed eight potential issues. The customers rated these problems on

Problem	Average rate
Test execution is slow on the developers machine	4.2
Hard to detect random failures	3.7
Hard to detect the cause of the failed tests	3.5
Hard to detect failures caused by a not-available or improperly working dependency	3.5
Test execution is slow on the build server	3.3
Hard to detect performance implications (speed, memory) of the changes	3.3
Hard to stop the integration test process in case of obvious general failures	3.2
Hard to integrate test execution (incl. reports) to the build server	2.9

Table 1. Questionnaire responses on test execution problems

a 1-5 scale, where 1 represented "not a problem" and 5 was "very painful". Table 1 shows the cumulated response sorted by the problem rating.

These responses showed two important facts:

1) It seems that all of the mentioned problems are valid issues at many companies (the lowest rate is around 3; 5% of the all individual rating was "1").

2) The top rated issues are the ones where the individual developer performance is directly impacted. With other words, these are the problems that force the *developers* to actively wait or spend time on issues that are not directly productive. This is probably due to the high cost factor of the development efforts in comparison to environmental costs (faster machine) or IT operational costs (expert who configures the build server).

In the following subsections, I will provide a more detailed list of problems categorized into four different groups.

6.1. Execution time

This is the most obvious problem encountered by the teams performing extensive automated functional/integration testing. These tests are much slower than unit tests. While a unit test can be executed in 50-100 milliseconds, integration tests run several seconds. Table 2 shows the execution statistics of three (anonymized) projects at TechTalk.

We have investigated the reasons behind the slow execution in different projects. In almost all of the projects, it turned out that the slow execution shared the same characteristics:

Project	**Test count**	**Execution time**	**Avg. time per test**
Project "T"	552	24 mins	2.6 secs
Project "L"	549	40 mins	4.4 secs
Project "R"	95	8 mins	5.1 secs

Table 2. Test execution times

1. The tests are not CPU intensive – the CPU on an average development machine runs on around 10% load

2. The preparation time (the "arrange" part) is usually bigger or similar to the execution time of the main testing action (the "act" part). The execution time of the verification ("assert") part was not significant.

3. Almost all of the tests communicated with at least one external component (the database), and in projects with UI automation, all tests also communicated with the web browser.

4. Only a few (<10%) of the tests used special external services (e.g. other than database, file system or web browser).

5. The external services (both common and special) were used exclusively by the test, i.e. the test was not prepared for sharing the service with other tests. For example, the test re-created the database before executing the action.

6. A lot of tests verified asynchronous components of the applications. These tests used mutexes, polling or timers to synchronize the verification.

Generally we can say that these tests are slow, because they are communicating with common external services, such as database, file system or web browser.

We have also investigated the behavior of the developers when interacting with such tests. This investigation was done through review discussions and pair programming. With this, we have identified the following behaviors:

1. The developers did not stop the test execution, even though it was obvious that there was an error. As a reason, they often stated that stopping the execution might leave the external systems in an inconsistent state, so the next execution would more likely provide false failures. This behavior can be observed on the developer machine (testing locally), but also on the build server, when there is a general error (e.g. connection to the database lost, all tests will fail).

2. The developers usually execute a larger set of tests than required for performing the verification of the component being developed. The common reason was that it was hard to overview the affected tests and select them by the testing tool.

Summarizing the dilemmas related to the slow test execution, the problem can be split into the following sub-problems:

1. Execution of a single tests is slow, because they communicate with common external services
2. More tests are executed than required because it is hard to define the tests to be executed
3. More tests are executed than required because the test execution cannot be stopped safely

6.2. Feedback about the execution

In the TDD-ATDD comparison (Section 5), I have outlined several differences (2, 3, 6, 7, 10 and 11) that are related to the necessity of the detailed feedback about the test execution. Because of the isolated and predictable nature of the unit tests, this feedback is not so important in TDD, hence the typical unit testing tools have no rich set of functionality in this area.

In many of the investigated projects, it was quite common to have transient or random test failures on the build server. This was usually caused by a temporarily unavailable external service or by a special defect in the application that causes the unpredictable error. Since the unit test tools execute every test once, the common practice was to re-run the entire test suite (even multiple times) in case of the suspicion of such transient error. Re-running the entire suite caused a significant delay in the investigation and fixing of the issue found.

In one project, the availability of some external dependencies was so low that it was very hard to achieve any test suite execution when all the tests passed (even if the application had no defect). The situation was hard enough itself, but it caused even more troubles as real test failures were frequently overseen because of the random defects. As a solution, the project team started to change the existing tests to fail with a special error when the problem was in the preparation ("arrange") phase. This change needed significant development efforts.

Another common problem was that tests reported trace information into different channels (console output and error, debug trace, etc.). The business

intentions (test steps) were also reported to the same channels. When investigating a test failure, it was a problem to see the merged trace information from the different sources on the one hand and separate them from the business intentions on the other hand.

As mentioned in the comparison, the tests may be verified and in some cases modified by the business and the testers. Also the project managers can use the test execution results for tracking the progress. Because the unit test tools were focused on providing feedback for the developers, the investigated projects applied additional reporting facilities to provide a “business-readable” report. This was some kind of HTML report published by the build server among other build artifacts. This configuration needed a fine-tuning of the build process for every new project.

Summarizing the difficulties related to the test execution feedback, the problem can be split into the following sub-problems:

1. Classification of failing tests to transient (“random”) failure, precondition failure and test failure.
2. Providing aggregated trace for the executed tests.
3. Presenting the execution log in a business-readable way.

6.3. Execution history

In fact the problems related to execution history are the sub-problems of the test execution feedback problem group. Since this is a bigger topic and there are a lot of possible ideas for improvements, I have decided to discuss it in a separate subsection.

For unit tests, the history of the test executions in the past is not too significant. Therefore the most common unit testing tools have no such feature. The TeamCity [30] build server product can collect and display a history about the test executions, but it is limited to the test result changes over the time. Many of the build server tools keep the previous execution to be able to run the previously failing tests first, but they cannot provide detailed statistics about the full execution history.

The problems or the possible improvements in this category are less concrete. This is mainly because there are no well-established practices what test execution based statistics are really useful for. In this subsection, I list some problems that can be addressed by collecting execution statistics.

The most common problem in this category is related to performance. Performance tests are costly to apply and it is hard to automate them in a way

that they can be regularly re-executed and the results can be compared. It is typical in software projects to perform performance tests and performance improvements in short campaigns. Such campaign can be regularly scheduled or triggered by a performance issue that appeared at the end users. To optimize this process, it is generally required to have some simple validation that can be regularly performed and can ensure that the changes in the code has no significant performance (execution time, memory usage, etc.) impact. In this regard, the automated functional tests could be used as a benchmark for the application's performance.

The test classification problem mentioned in the previous subsection has also an aspect for the execution history. For identifying some issue categories, the execution history gives good input. For example, the difference between the transient errors caused by a temporarily unavailable dependency can be better distinguished from a "random failure" based on the execution history. In some cases, the history can help to identify and solve the issues as well (e.g. at 2 am some tests usually fail – maybe they interfere with the daily backup process).

The test execution history can be also used for giving earlier feedback about the more risky tests (recently failed or newly added).

Finally, collecting execution statistics can help the developers to identify the unstable areas of the application that frequently fail after changes in the code.

Summarizing the problems related to the test execution history, the problem can be split into the following sub-problems:

1. Using the automated functional tests as performance benchmarks
2. Classification of tests based on the execution history
3. Give earlier feedback about the more risky tests (recently failed or newly added)

6.4. Test setup

Unit tests are isolated and there is no need for any external dependencies to be setup. As opposed to them, in the case of ATDD tests, it is quite common that some test set up tasks have to be performed in prior to the test execution.

In all of the investigated projects there were special test setup tasks required for running ATDD tests: deploying a test instance of the application, copy file resources, start web server, create database. This setup tasks were either performed by the build process or they were done in the "setup" phase of the unit test execution.

In projects, where the setup tasks were fulfilled in the build process, the developers had to do extra work for performing ad-hoc tests (i.e. pick and run an individual test or a few tests), because the ad-hoc testing facilities bypassed the standard build process.

However, in projects where the setup tasks were built up from code in the “setup” phase, the configuration and maintenance costs of this code were difficult, because the general-purpose programming languages (in our case C#) were not suitable for setup tasks.

The analysis has shown that the root causes in these scenarios are the following:

1. Test setup tasks are not bound to the test execution.
2. Test setup tasks have to be described in general-purpose language.
3. Different configuration sets for executing tests (e.g locally or on the server) cannot be defined in one place.

7. SpecRun

As mentioned earlier, we saw that many of the testing efficiency issues are caused by the tool support. Therefore in June 2011 TechTalk decided to launch a new tool, called SpecRun [17] to provide solutions for some of these problems. In November 2011, SpecRun was in beta phase and we have been collecting feedback about it. The final release will be announced soon as a commercial product, but with free license for open source and non-commercial projects.

8. Improving execution efficiency

We have investigated several ideas to address the test execution efficiency issues outlined in Section 6. In some areas we were able to prototype and measure the result of these ideas. In other cases the idea was only described but not implemented and verified yet. This section provides a summary of these improvements, categorized by the problem groups described in Section 6.

When we considered improvements in testing efficiency, we wanted to find solutions that need no or minimal change in the existing tests. Obviously, if

these improvements are used in combination with good test automation practices, the benefits can be even more improved. In the investigated projects, however, the design and coding quality of the tests were fairly good and it was not possible to improve it significantly with reasonable efforts.

8.1. Execution time

With regard to the execution time we have identified three different solution areas: parallel execution, gentle test execution termination and the execution of impacted tests. This subsection describes these areas and their impact wherever possible.

Parallel execution. Parallel execution is an obvious solution for faster test execution. The key point in this area was to realize that the test execution time is mainly caused by the communication with external services. I have investigated and proven that the execution time of these tests can be significantly decreased even on a single machine. Though the development machines we tested were all multi-core, the analysis of the CPU utilization showed that the performance improvement was caused by the test characteristics mentioned in Section 6.1. Just to understand the improvement, we can imagine the parallel execution in a way that one test-thread is waiting for the external service to respond (e.g. I/O), while the other could perform the CPU intensive calculations and vica versa. This way the resources can be utilized in a more balanced way.

On the bases of the measurements, the optimal degree of parallelism in the investigated applications was around 3 test execution threads, where about 50-60% performance improvement was measurable. The execution times with different thread count for project “R” is shown in Table 3.

Thread count	Avg. CPU load	Execution time	Change
1	30%	4:27 mins	0%
2	50%	2:34 mins	42%
3	80%	2:05 mins	53%
4	100%	1:47 mins	60%
5	100%	2:01 mins	55%

Table 3. Test execution times of project “R” on different thread count

As our goal was to keep the test unchanged, we had to solve the problem of exclusive access of the external services, when the parallel execution was introduced. For example, if a test exclusively used the database, we had to make “clones” of the database and ensure that the tests in the different threads use

different instances. This has been achieved with the help of special test setup configurations that could refer to a variable “TestThreadId”. This variable contains the zero-based index of the current test-thread.

For the common external services, like database or file system, it was possible to create two or three clones. However, for some special services this was sometimes impossible. In project “R”, there was a server component that could only work in a one-instance-per-machine style. The project “L” ran tests that connected to a Microsoft Team Foundation Server [31], where creating many test projects was not convenient. In the investigated projects, the number of tests using such special service was below 10%, so instead of making them parallelizable we simply provided an option to exclude them from parallel execution, meaning that they were always “bound” to a specific test execution thread.

For the parallel execution the isolation level of the different test execution threads was another issue. In the .NET environment, where the tool runs, there are basically two options for isolation. The test threads can be executed in different Application Domains [32] or in different processes. The first provides a better performance (less overhead), the other provides better isolation. Finally, we decided to implement the AppDomain isolation first, which seemed to be sufficient for the majority of the applications. We saw an application however, where the external service was a native component that allocated per-process resources (a notification message broker). For this project the AppDomain isolation did not provide the expected result: the execution time did not increase by increasing the thread count as shown in Table 4. Such projects need to be supported by providing process-level isolation in the future.

Test thread count	**Total execution time (secs)**
1	134
2	113
3	179

Table 4. Test execution times of a project that cannot be parallelized with AppDomain isolation

We have also investigated the parallel execution on different machines. While in some special cases this would be also beneficent, in the majority of the projects we saw the costs of setting up and maintaining extra machines were too high.

Another challenge was to keep the overhead of parallel execution at minimum. Based on the measurements, the final solution built into SpecRun has less than 5% overhead at 3 parallel threads.

Gentle test execution termination. As mentioned in Section 6.1, it was problematic to stop the test execution in such a gentle way that no external services were left in an inconsistent state. This problem was observable on the developer machine and also on the build server.

On the developer machine we implemented a simple solution that responded to the Ctrl+C keystroke and stopped the execution after the currently running test(s) had been finished. This simple feature became very popular among the developers, especially together with the adaptive test order (see Section 8.3).

On the build server, we aimed to find a solution that works without any user interaction. After investigating several possibilities, we decided to test another simple solution. The test execution can be configured to stop after a specific number (e.g. 10) of failed tests. We verified the usefulness of this feature by letting the individual project teams consider whether they use it or not. After a few months of test period, all the teams still used this feature and reported on occasions when it saved significant time. This is typical when a long-running build process starts, but some general and obvious errors (e.g. external service has not been started) would cause the majority of the tests to fail.

Execution of impacted tests. Deciding efficiently what tests are impacted by a code change is a complex problem. There are ways to gather such information. These are either done by static code analysis or by test impact analysis of the previous execution. The problem with the static code analysis is that it cannot handle dynamic invocations (quite common in .NET), so it cannot provide a trustful result. The execution analysis can be done through the .NET profiling API (see also [29]), but this leads to a significant performance decrease (almost doubles the execution time), so it cannot be efficiently used in our target domain (developer machines). Defining better solutions for this area is one of the most interesting topics of further research.

Currently, we have solved this problem by improving the internal project development guidelines. First of all, we emphasized that by the nature of these tests, it is a valid scenario if an integration error is only caught by the build server (i.e. someone "breaks the build") (see difference 8 in Section 5). We asked the developers to perform a reasonable set of checks before committing their code to the source control system. We defined the "reasonable set" by tests that belong to the current iteration. These tests were specially tagged (e.g. "current_iteration") for an easier execution.

8.2. Feedback about the execution

In this problem group, we were able to provide well usable solutions for many of the mentioned problems. Therefore, this subsection contains only a brief summary about these.

In order to be able to present the execution log in a business-readable way, we have integrated the HTML report generation as a first class citizen into SpecRun. The generated HTML reports contain the detailed test execution traces, but also summary sections (e.g. about failing tests) and a graph for visualizing parallel test execution and performance indicators. Figure 3 shows a generated HTML report.

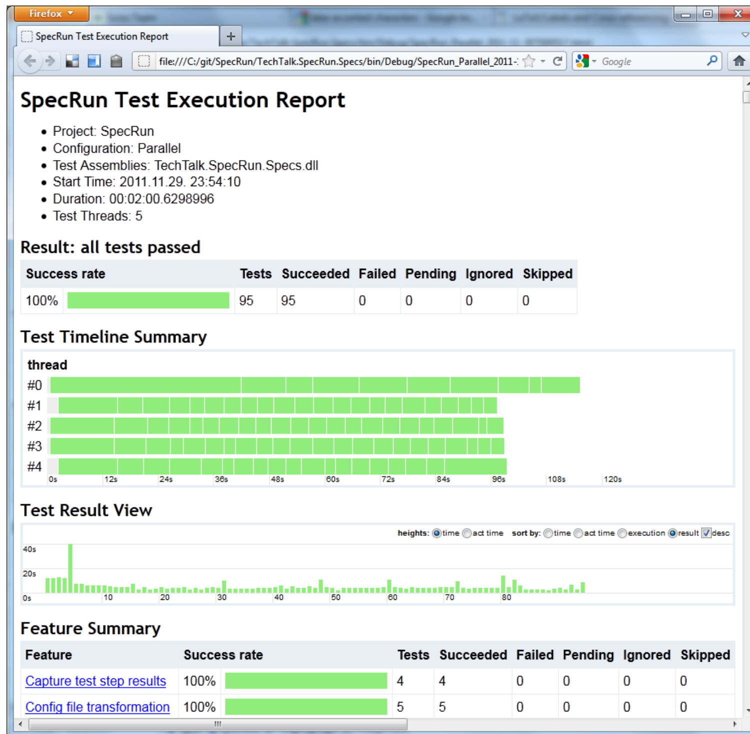

Figure 3. Test execution HTML report

In order to provide aggregated trace, we introduced two trace channels: the business trace and the technical trace. The trace information provided by the tests were redirected to one of these two traces depending on the fact if it was about describing the business intention (test steps) or something else. To capture the business intention messages, a special Listener extension was made for SpecFlow. Both of these channels are displayed in the report split by the test steps. Additional timing and result information were provided for each individual step.

The most complex solution was to address the transient or random failures. For this purpose a special retry mechanism was implemented. For the retry, it

has to be specified what should trigger the retry (failing tests, always, never, history-based heuristic) and the retry count. (The history-based triggering is not implemented yet.) Regarding the efficiency of the retry mechanism, the feedback is not so clear. None of the projects used the “always” option for retry. With the “failing test” triggering option, the transient errors (external service temporarily unavailable) can be certainly caught, however, the random failures can be detected only randomly (when the first execution fails). In addition to this, the retry function was sometimes annoying when executed locally, therefore, it was usually turned off for the local execution profile.

8.3. Execution history

Since this problem group provides the most long-term potential for efficiency improvements, we have decided to establish the collection of test execution statistics right from the beginning.

There are a lot of interesting techniques for test case prioritization based on source code analysis and/or execution history (e.g. [15, 33, 34, 35]). This topic itself is a big research field on its own. My initial goal was to set up an infrastructure that allows implementing such ideas later.

SpecRun has a server component (optional) that can collect test execution results and can provide statistics for other tools. The SpecRun server uses the CQRS [36] architecture that enables to process incoming test results asynchronously with minimal overhead on the caller side.

To be able to provide execution statistics to many different (even 3rd party) tools, the server uses REST-based OData [37] interface over HTTP protocol.

Currently, the server collects the following information about the test executions:

1. Test result
2. Test execution time
3. Execution time of the “act” part of the test
4. Environment information (machine, testing profile)

The server can calculate a cumulated test status value from the last 10 executions. This cumulated status provides more information than the simple pass/fail pair; for instance, “recovering” denotes a test that has failed in the past but there have been a few successful executions since the last failure.

SpecRun, as a client for the SpecRun server, can use the execution statistics for deciding the test execution order. It starts the recently failing and new

tests first. This simple heuristic can be later replaced by a more sophisticated solution based on the studies referred before.

Although the server already collects the information for execution time benchmarking, this has not been implemented yet. For a useful benchmarking result, the data have to be cleaned from extreme deviations (e.g. test executed in the test suite first is usually much slower) and have to be normalized (different execution machines might have different performance). This area is another important topic of further research.

8.4. Test setup

To address the problems mentioned in the test setup category, we have introduced two concepts in SpecRun.

All configurations related to the test execution are grouped into one XML file, called test profile. Different test profiles can be specified for the different testing scenarios (e.g. running test of the current iteration locally; running a full regression test on the build server).

The second concept is the test deployment step. In the testing profiles, one or more test deployment steps can be defined. These steps can be either global or local to a test execution thread. The set of possible steps is extensible, however, some common steps (relocate, copy folder, start IIS express, change configuration) have been built-in.

Both of these concepts were received well in the projects.

9. Conclusion and future work

The goal of this paper was to collect possibilities for improving testing efficiency by providing specialized execution environment for automated functional tests developed through the ATDD technique.

First, Sections 2-5 described the current testing practices for TDD and ATDD, furthermore their key differences. These differences lead us to a deeper investigation of the potential problems caused by using TDD tools for executing ATDD tests. These problems are described and categorised into four groups in Section 6.

To address these problems and prototype potential solutions, TechTalk has started a new product called SpecRun. Section 7 briefly summarizes the current status of the tool.

Finally, Section 8 describes concrete ideas and implemented solutions that address the outlined problems.

As a conclusion, we can say that with a specialized test execution environment, testing efficiency can be significantly improved without changing the existing test automation practices. Regularly executed tests that are bound to functional elements of the applications provide a very good source of information that can be used to further improve the efficiency. I described performance benchmarking as one of such, which is currently being researched.

Providing an efficient way of deciding which tests should be executed for a concrete code change is another area of further research.

Acknowledgment. I would like to thank to my colleagues at TechTalk (especially Jonas Bandi and Christian Hassa) for collecting these experiences. The results are partly based on the discussions I had with them and with others. These discussions were partly published in [38] and [39]. Many thanks to my wife, Adrienn Kolláth for the support and splitting up the long sentences. For the encouragement, I would like to thank to Dr. László Kozma and Dr. Sándor Sike from Eötvös Loránd University, Budapest, Faculty of Informatics.

References

[1] **Beck, K.,** *Test-Driven Development by Example*, Addison Wesley, 2003.

[2] Behavior Driven Development, Wikipedia.
`http://en.wikipedia.org/wiki/Behavior_Driven_Development`

[3] **North, D.,** *How to sell BDD to the business, Agile Specifications*, BDD and Testing eXchange, 2009. `http://bit.ly/4wWuQh`

[4] **Koskela, L.,** *Test Driven: Practical TDD and Acceptance TDD for Java Developers*, Manning, 2007.

[5] TechTalk. `http://www.techtalk.at`

[6] SpecFlow. `http://www.specflow.org`

[7] NUnit. `http://nunit.org/`

[8] Verifying Code by Using Unit Tests, MSDN.
`http://msdn.microsoft.com/en-us/library/dd264975.aspx`

[9] **Bertolino, A.,** Software testing research: achievements, challenges, dreams, in: *Proceedings of the 2007 Future of Software Engineering IEEE Computer Society*, 2007, pp. 85–103.

[10] **Harrold, M.,** Testing: A Roadmap, *International Conference on Software Engineering*, Limerick, Ireland, 2000, pp. 61–72.

[11] **Jones, W.D., J.P. Hudepohl, T.M. Khoshgoftaar and E.B. Allen,** Application of a usage profile in software quality models, in: *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, IEEE Computer Society, Amsterdam, Netherlands, 1999, pp. 148–157.

[12] **Khoshgoftaar, T.M., E.B. Allen, W.D. Jones and J.P. Hudepohl,** Accuracy of software quality models over multiple releases, *Annals of Software Engineering*, **9** (2000), 103–116.

[13] **Kouchakdjian, A. and R. Fietkiewicz,** Improving a product with usage-based testing, *Information and Software Technology*, **42(12)** (2000), 809–814.

[14] **Kuhn, D.R.,** Fault classes and error detection capability of specification-based testing, *ACM Trans. Softw. Eng. Methodol.*, **8(4)** (1999), 411–424.

[15] **Rothermel, G., R.H. Untch, Chengyun Chu and M.J. Harrold,** Prioritizing test cases for regression testing, *Software Engineering, IEEE Transactions on*, **27(10)** (2001), 929–948.

[16] **Chen, T., H. Leung and I. Mak,** Adaptive Random Testing, *Lecture Notes in Computer Science*, **3321** (2005), 3156–3157.

[17] SpecRun. `http://www.specrun.com`

[18] **Fowler, M.,** *Specification by Example*, Martin Fowler's Bliki, `http://www.martinfowler.com/bliki/SpecificationByExample.html`

[19] **Adzic, G.,** *Specification by Example: How Successful Teams Deliver the Right Software*, Manning, 2011.

[20] **Park, S. and F. Maurer,** *A Literature Review on Story Test Driven Development*, in: XP'2010, Trondheim, Norway, 2010, 208–213.

[21] **Melnik, G.,** *Empirical Analyses of Executable Acceptance Test Driven Development*, University of Calgary, PhD Thesis, 2007.

[22] User Acceptance Testing, Wikipedia. `http://bit.ly/t3lZ8N`

[23] **Fowler, M.,** *Refactoring Home Page*, `http://www.refactoring.com/`

[24] Arrange Act Assert, `http://c2.com/cgi-bin/wiki?ArrangeActAssert`

[25] Test Case Explosion, Software Test Glossary, `http://www.zeta-test.com/glossary-t.html#a2032`

[26] **Beck, K.,** *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[27] Scrum Acceptance Criteria, Scrum Methodology, `http://scrummethodology.com/scrum-acceptance-criteria/`

[28] Happy Path, Wikipedia. `http://en.wikipedia.org/wiki/Happy_path`

[29] **Gousset, M.,** Test impact analysis in Visual Studio 2010, *Visual Studio Magazine*, 2011, `http://bit.ly/vQ7sUd`

[30] TeamCity, `http://www.jetbrains.com/teamcity/`

[31] Microsoft Team Foundation Server 2010,
http://msdn.microsoft.com/en-us/vstudio/ff637362

[32] Application Domains, MSDN,
http://msdn.microsoft.com/en-us/library/2bh4z9hs.aspx

[33] **Elbaum, S., A.G. Malishevsky and G. Rothermel,** Test case prioritization: a family of empirical studies, *IEEE Transactions on Software Engineering*, **28(2)** (2002), 159–182.

[34] **Park, H., H. Ryu and J. Baik,** Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing, in: *Proceedings of the 2nd International Conference on Secure System Integration and Reliability Improvement (SSIRI 2008)*, Yokohama, Japan, 2008, pp. 39–46.

[35] **Zengkai, M. and Z. Jianjun,** Test case prioritization based on analysis of program structure, in: *Proceeding of 15th IEEE Asia-Pacific Software Engineering Confer-ence (APSEC 2008)*, Beijing, China, 2008, pp. 471–478.

[36] **Fowler, M.,** *CQRS*, Martin Fowler's Bliki, July 2011,
http://martinfowler.com/bliki/CQRS.html

[37] Open Data Protocol, http://www.odata.org/

[38] **Bandi, J., C. Hassa and G. Nagy,** *Using SpecFlow for BDD ATDD and (U)TDD?*, SpecFlow Forum, 2010, http://bit.ly/vT0scw

[39] **Bandi, J.,** *Classifying BDD Tools (Unit-Test-Driven vs. Acceptance Test Driven) and a bit of BDD history*, 2010, http://bit.ly/ajT9m5

G. Nagy
Department of Software Technology and Methodology
Faculty of Informatics
Eötvös Loránd University
H-1117 Budapest, Pázmány P. sétány 1/C
Hungary
gaspar.nagy@gmail.com