Delft University of Technology
Software Engineering Research Group
Technical Report Series

Aiding Software Developers to Maintain Developer Tests

Victor Hurdugaci, Andy Zaidman

Report TUD-SERG-2012-002

TUD-SERG-2012-002

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication at the 2012 European Conference on Software Maintenance and Reengineering (CSMR)

Aiding Software Developers to Maintain Developer Tests

Victor Hurdugaci
Delft University of Technology
The Netherlands
Email: contact@victorhurdugaci.com

Andy Zaidman
Delft University of Technology
The Netherlands
Email: a.e.zaidman@tudelft.nl

***Abstract*—Unit and integration tests can be invaluable during software maintenance as they help to understand pieces of code, they help with quality assurance and they build up confidence amongst developers. Unfortunately then, previous research has shown that unit tests do not always co-evolve nicely with the production code, thus leaving the software vulnerable. This paper presents TestNForce, a tool that helps developers to identify the unit tests that need to be altered and executed after a code change, thereby reducing the effort needed to keep the unit tests in sync with the changes to the production code. In order to evaluate TestNForce, we perform a user study that evaluates the adequacy, usefulness and completeness of TestNForce*.**

I. Introduction

Lehman has taught us that a software system must evolve, or it becomes progressively less useful [2]. When evolving software, the source code is the main artifact typically considered, as this concept stands central when thinking of software. Software, however, is multi-dimensional, and so is the development process behind it. This multi-dimensionality lies in the fact that to develop high-quality source code, other artifacts are needed, e.g., requirements, documentation, tests, etc. [3]. A software development process aiming for quality should thus allow these artifacts to co-evolve gracefully alongside their respective dimensions.

One artifact which is of primary importance when developing high-quality software, is the so-called developer test, i.e., a codified unit or integration test written by developers [4]. Indeed, a 2002 report from the NIST indicates that catching defects early during (unit) testing lowers the total development cost significantly [5].

Intuitively, we know that ideally, the *production code* and *test code* should co-evolve, not in the least to have a permanent safety net during reengineering [6]. A previous study [7], however, has shown that not all software projects uphold a process whereby graceful co-evolution of production code and test code takes place. This effectively means that the software is vulnerable for extended periods of time as the production code evolves, but the test code does not follow (immediately). In this context, Moonen et al. have shown that even while refactorings are behaviour preserving, they potentially invalidate tests [8]. In the same vein, Elbaum et al. concluded that even minor changes in production code can significantly affect test coverage [9].

The reasons for this lack of co-evolution are manifold:

- lack of time or resources because the maintenance of a test suite is very costly [10], [11], [12]
- lack of awareness of the existence of tests for a particular functionality (e.g., due to lack of traceability and/or bad naming conventions) [13]
- lack of enough knowledge of the software system or lack of tool-support to identify all covering tests [14], typically resulting in either too few or too many tests being executed [11]
- lack of time to run the tests (running the unit test suite may take from seconds to hours [11])

While we cannot solve all of these issues, tool-support that creates awareness and supports software developers in identifying and remembering which tests *cover* a particular piece of source code can alleviate these issues. It is in this context that we have developed TestNForce[1], a plug-in for Microsoft Visual Studio 2010 that allows developers to see which tests cover a piece of (changed) source code. Additionally, it helps the developer in remembering that the test code should be adjusted before committing source code to version control.

This paper introduces TestNForce and addresses the following research questions:

RQ1 Can such a tool be built with acceptable performance?
RQ2 Is the tool considered useful by developers?
RQ3 Do the developers experience the tool as a hindrance during development?

This paper is structured as follows: in Section II we first present a number of usage scenarios for TestNForce, after which we give a description of the inner-workings of the tool. Section III describes our experimental setup, while Section IV presents the results of the experiment. In Section V we discuss our findings and we present threats to validity. Section VI introduces related work and Section VII concludes this paper.

*This work is described in more detail in the MSc thesis of Victor Hurdugaci [1].

[1]A video of the use of TestNForce and the source code of the tool are available at http://swerl.tudelft.nl/bin/view/Main/TestNForce

II. Overview of TestNForce

TestNForce is a plug-in for Microsoft Visual Studio 2010. Its main aim is to assist software developers in determining which subset of unit tests should be changed and/or executed after adapting the production code. Section II-A first introduces common usage scenarios for TestNForce, after which Sections II-B and II-C detail the internals of our implementation.

A. Usage scenarios of TestNForce

While creating TestNForce, we had three primary scenarios in mind to help software engineers in maintaining their developer tests. We now discuss the three scenarios:

Scenario 1: Show covering tests

Motivation: A software developer might be interested in knowing which tests cover a particular piece of code, (1) because the test code might explain how the piece of code works, in particular, how parameters should be initialized, (2) to understand the impact of his changes on the tests and (3) to ensure that a safety net exists for that particular piece of code before starting to change it.
Instantiation: When right clicking on a piece of code, a context menu appears from which you can choose the "Show covering tests" option (see Figure 1). If the developer activates this option for a line of code, all test methods that cover the surrounding method will be listed. If more than one method is selected while right clicking, all test methods that cover the selected methods are shown. Figure 2 shows how TestNForce presents the results to the user.

Scenario 2: What tests do I need to run?

Motivation: After adjusting the production code the developer wants to know which tests should be investigated and possibly changed. While this might seem like a trivial task, this is often not the case. In particular, when the typical naming convention between production and test code (a class `string` is tested by a class `stringTest`) is missing and/or when a more integration test oriented style of testing is used, it becomes hard to manually trace the covering test methods.
Instantiation: In the test menu (right upper corner in Figure 1) TestNForce introduces an extra menu item "What tests do I need to run?". This option analyses all the files in the Visual Studio solution and returns a list of test methods that cover units of production code that have been changed.

Scenario 3: Enforcing self-contained commits

Motivation: A developer might sometimes forget that changes to production code do often need to be followed up by changes to test code as well. While these follow-up changes are not always necessary previous research by, e.g., Moonen et al., has shown that a number of refactorings in production code necessitate changes to test code as well [8]. We want to create awareness with the developer that while he changed production code, (some of) the covering tests were not changed.
Instantiation: In order to make sure that this co-evolution happens immediately, TestNForce has a commit policy. This commit policy ensures self-contained commits [15], i.e., commits whereby changes to production *and* test code are committed simultaneously. Just before committing the changes (to production code), TestNForce will determine whether the tests that cover the production code have been changed as well. Figure 3 shows the warning that TestNforce provides when the mapping index between production and test code is not up to date.

B. Building the TestNForce production/test code index

In order to provide the functionality for helping software developers in the scenarios discussed above, TestNForce

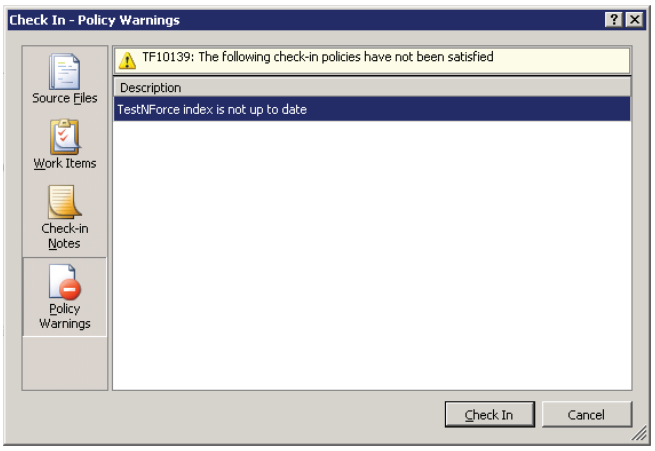

Figure 3. The TestNForce commit policy shows a warning that the mapping index is not up to date

follows a seven step plan to build an index that maps a production code method to its covering test(s). This seven step plan is shown in Figure 4. We will now go over each of these seven steps in more detail.

Step 1: Solution meta identifier: In Visual Studio, a solution is a structure that contains a group of one or more projects. These projects work together to create an application. The first step analyses the Visual Studio solution and stores information about it. In particular, we want to know which projects reside in the Visual Studio solution and their respective paths.

Step 2: Project meta identifier: The second step is meant to determine whether each of the projects that were identified in the previous step can be used by TestNForce. The projects that can be used by TestNForce are C# (test) projects.

Step 3: Build: In this stage the project is built. If the build fails, the TestNForce analysis stops.

Step 4: Instrumentation: The binaries that we obtained from the previous step are now instrumented so that their execution can be traced. This process is very similar to what a typical code coverage tool does. In terms of implementation, we were able to reuse `vsinstr`, a tool provided by the .NET Framework, which facilitates the instrumentation of .NET binaries.

Step 5: Identify tests: We now use .NET Reflection to identify tests in each test assembly. In particular, from `mstest`'s[2] point of view, a test method is a public instance method with no arguments decorated with the `TestMethodAttribute` attribute.

Step 6: Run tests: In this step we need to determine which parts of the production code are covered by individual test cases. In order to obtain this information, we followed an approach similar to Galli et al. [16]. In particular, we execute each test case individually and use standard tools

[2]MSTest is a command line utility from Microsoft that executes unit tests created in Visual Studio.

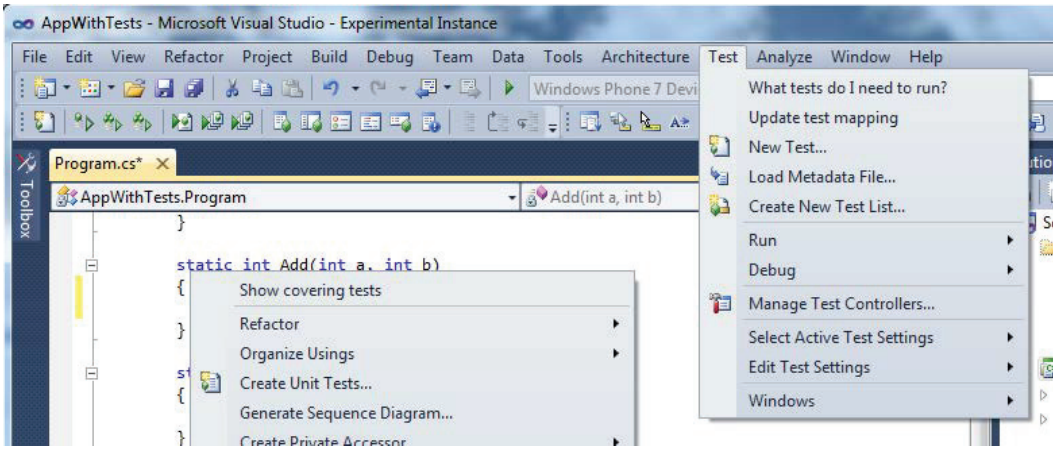

Figure 1. Additional TestNForce menus in Visual Studio

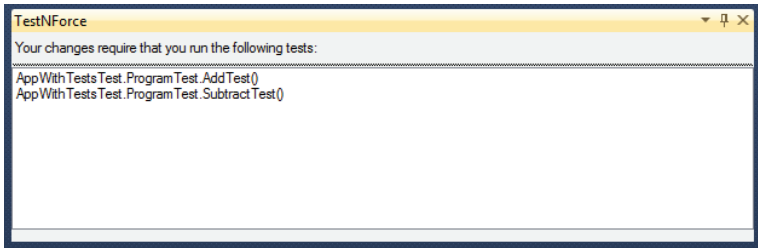

Figure 2. Result of asking TestNForce to determine which tests cover a particular piece of code

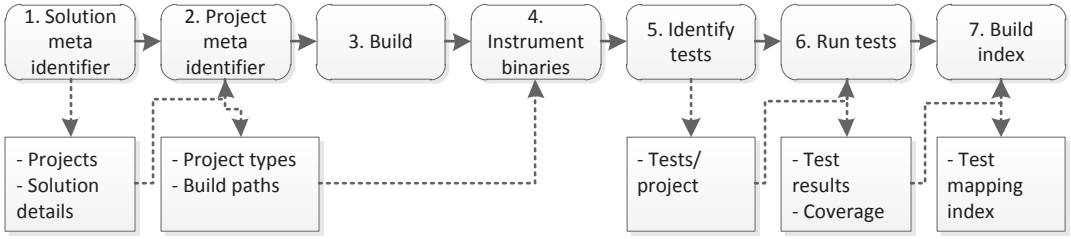

Figure 4. Seven steps of the TestNForce process

like `vsinstr` and `perfmon` to track down the execution path of individual test cases.

Step 7: Build index: Using the execution paths of individual test cases (see Step 6), we now build the test mapping index. This index keeps a many-to-many mapping between test methods and the tested methods. The many-to-many relation is needed because a test can cover, and usually does, multiple methods while a method can be covered by multiple tests. The index is stored next to the Visual Studio solution file in XML format. For space saving reasons, each indexed method gets an Id and the mapping section of the file refers to their Ids. Below is a part of a mapping file:

```
<Methods>
  <Method Name="P.Add(System.Int32,System.Int32)"
                ID="0" />
  <Method Name="P.AddTest()" ID="1" />
  <Method Name="P.Cmp(System.Int32)" ID="2" />
</Methods>
<TestMapping>
  <TestMap MethodID="0" TestID="1" />
  <TestMap MethodID="2" TestID="1" />
</TestMapping>
```

C. Code comparison

After the index has been built, TestNForce needs to establish whether any code has changed. A first rudimentary check is to see whether the file containing a unit of code has changed based on the date and time of change. This check, however, is still very coarse grained. In order to provide the developer with a more fine-grained view of what has changed, we decided to compare the old and the current versions of units of code. In particular, we are checking whether the two pieces of code are equivalent except for changes to layout and/or addition or removal of comments.

III. Experimental Setup

In our experiment we want to verify whether TestNForce is able to change the software development and testing habits of its users. Specifically, we are interested in the opinion of developers regarding the *adequacy*, *usability* and

completeness of TestNForce. For this, we let eight experimental subjects work with TestNForce during a number of programming assignments. We employ a one-group pretest-posttest pre-experimental design [17].

A. One-group pretest-posttest

In a one-group pretest-posttest pre-experimental design, only one group is tested. Instead of having a control group like in a typical controlled experiment, the experimental group is subjected to an extra test before the experiment is conducted. This test serves as a baseline to which the measurements gathered after the experiment can be compared. During pretesting and posttesting, the subjects are measured in terms of the dependent variables. Usually, the same questionnaire is used both before and after varying the independent variable (i.e., introducing the tool). By using the same questions, the pretest and posttest results can be compared more easily. The fact that no control group is present is why this type of experiment is called pre-experimental, in particular because the lack of the control group hinders us to identify an event related to the dependent variable that intervenes between the pretest and the posttest where the effects could be confused with those of the independent variable. Nevertheless the one-group pretest-posttest design allows us to report on facts of real user-behaviour, even those observed in limited-sample experiences [18].

For our questionnaires we employ close-end matrix questions in which respondents can rate a number of statements on a 1 to 5 scale, ranging from strongly disagree to strongly agree (the so-called Likert scale).

Pretest design: For the pretest[3] a total number of four themes were chosen. Each theme relates to a different aspect of the experiment. Most themes are intended to determine possible external variables that might influence the dependent variables, other than the independent variable that is being examined (i.e., the use of TestNForce). An excerpt of the pretest questionnaire can be seen in Table I.

1) Participants background (questions 1a-1d). Questions about age, profession and education.
2) Development experience (questions 2a-2i). Subjects were asked to rate their development skills, experience with C# and Visual Studio.
3) Testing experience (questions 3a-3h). Information about the testing skills was collected using the questions in this category.
4) Expectations from TestNForce (questions 4a-4f). The participants read an abstract description of TestNForce (printed below) and expressed their expectations with regard to the usefulness of such a tool.

Posttest design: After the subjects have completed their assignments, they have to fill out a second questionnaire serving as a posttest, of which the primary intent is to measure whether the subjects expectations with regard to a unit testing aid like TestNForce are met.

[3]The pre- and posttest design are shown in detail in [1].

2a	I consider myself an experienced developer
2b	What is (are) the development environment(s) that you are experienced with? [open question]
2c	I consider myself an experienced Visual Studio user
2d	What is (are) the programming language(s) that you are experienced with? [open question]
2e	I consider myself experienced with a .NET language
2f	I consider myself an experienced C# programmer
2g	I worked before on large scale software projects
2h	I understand the challenges that arise in software projects
2i	I consider myself familiar with Jurassic
3a	I consider myself an experienced tester
3b	I write tests for most of the code I am writing
3c	What kind of tests you did in the past? [open question]
3d	I believe that is better to deliver fast, with possible defects than to spend extra time on testing
3e	I believe that one-man projects dont require automated tests
3f	Is it common that the size of the test code to be greater than the size of the tested code
3g	I believe that the amount of resources spent on developing and maintaining test code can be greater than those spent for the tested code
3h	How much code coverage is considered “good”? [open question]
Statement	With a test impact tool, one should be able to decide what tests to run after changing the code. In other words, the tool will provide a list of tests that are relevant for the change. Such a tool will inform the developer about the tests that cover the code she/he changed. Furthermore, upon check-in (commit) to the version control system, the tool will prevent this action if tests corresponding to the changed code were not executed and, optionally, updated.
4a	I think that I would use such a tool
4b	I think that such a tool reduces testing time
4c	I think that such a tool reduces the overall development time
4d	I think that such a tool might be annoying
4e	I think that such a tool is solving a real problem

Table I
EXCERPT FROM THE PRETEST. UNLESS INDICATED OTHERWISE, ALL QUESTIONS ARE TO BE ANSWERED ON A LIKERT SCALE FROM “STRONGLY DISAGREE” TO “STRONGLY AGREE”.

In the posttest, a number of different issues are addressed. We have subdivided the questions into seven categories:

1) Questions about TestNForce (questions 1a-1c). Did the subjects find TestNForce useful and easy to use.
2) TestNForce in relation to the assignment (questions 2a-2c). Did the tool help to solve the assignment.
3) TestNForce and Team Foundation Server (questions 3a-3d). Is the integration between both useful and usable.
4) Usability (questions 4a-4f). General usability questions on TestNForce.
5) What is missing (questions 5a-5f). A number of ideas that we have for future versions of TestNForce are listed here.
6) Assignment (questions 6a-6e). Was the assignment too difficult, was there enough time?
7) Experiment (questions 7a-7f). Did the experiment have the right focus, was the case study appropriate?

B. Assignment

In order to confront our experimental subjects with TestNForce, we created a number of programming assignments that would require them to use TestNForce.

Project. The assignment had to be created around a project. For the selection of a candidate project, we set forward a number of requirements:

- It should not be a trivial application, yet easy to grasp in a short period of time.
- It should have a considerable number of test cases that pass.
- The available unit tests should have a good level of test coverage.
- Considering the tooling infrastructure, the project should be written entirely in C#.
- Ideally, the test cases would need more than a few minutes to run.
- Ideally, the project should not be known by any of the subjects to create a level playing field.

We started by investigating CodePlex[4], the biggest community site hosting open source .NET projects. Taking our requirements into consideration, we finally settled on a project called *Jurassic*[5]. Jurassic is an implementation of the ECMAScript language and runtime[6]. Worth mentioning for Jurassic are the facts that the project has 344 test cases available, is reasonably complex, but is still easy to learn due to its well-designed architecture.

Tasks. The assignment is constructed around a scenario that involves a number of programming tasks. We now describe the tasks, for all details we refer to [1][p. 84]:

1) The first task is asking the subject to figure out which test cases are covering the method `HasVariable` of the `MethodOptimizationHints.cs` class. First, the subject is asked to perform this task without TestNForce and once he is reasonably sure of his results, the subject can use TestNForce to see which tests are covering the method.
2) The second task is about change risk. The subject has to decide if adding a new base type to the compiler core breaks any tests. TestNForce can be used after identifying what methods need to be changed either by invoking the "What tests cover this method?" option or by actually making the changed and invoking "What tests should I run".
3) The third task is to fix a method that is causing some tests to fail. The participants were not told which tests are covering that method so they have two options (1) either execute all tests, which it not feasible because it takes 40 minutes to run them or (2) use TestNForce to identify the tests. Then, armed with the tests cases, they have to proceed and change the method. Finally, they have to prove that the change is good by invoking the (two) covering tests.

4&5) The fourth and fifth assignment can be done together. The fourth assignment asks to check in the changes that the subject did. This is not possible until the index is updated and added to the project (the fifth assignment). However, updating the index takes 28 minutes so, participants were asked to start the process, but not wait until it finished.

[4]http://codeplex.com, last visited September 1st, 2011.
[5]http://jurassic.codeplex.com/
[6]http://jurassic.codeplex.com/

C. Pilot run

After the design of the experiment was completed, but before the actual experiment run, we conducted a pilot run of the experiment. Such a run was scheduled in order to be proactive and catch any unexpected issues.

The pilot run revealed two software related issues. One was a bug in TestNForce, which caused the index not to be updated correctly. The other issue was related to a configuration issue of the Team Foundation Server that we used for pilot run.

Additionally, we also made a number of modifications to the experiment itself, in particular:

- We clarified the first assignment, in particular, we added a hint to the assignment to use the *Find References* feature of Visual Studio.
- We added a short introduction to the Visual Studio environment to accommodate people with experience with other IDEs.

IV. Experiment

Based on the experimental setup that we described in Section III we conducted an experiment involving eight subjects. This section describes the details of the experiment and the results.

A. Subject profile

The eight volunteers that participated in our experiment were recruited within the computer science faculty of the Delft University of Technology. They all had a computer science background and they all had either an MSc degree, PhD degree or were very close to one. Furthermore, all participants were male and had ages between 23 and 27. A number of the participants had obtained either their BSc or MSc degree at a university other than the Delft University of Technology, indicating a diverse background of the participants. Figure 5 provides more details on the participants.

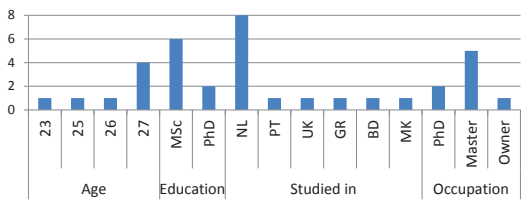

Figure 5. Subject profiles

B. Pretest

1) General inquiry: All participants consider themselves at least averagely experienced developers. Question 2a of Figure 6 shows that all participants attributed themselves a score of 3 or 4, with the median at 4[7]. With no outliers, the group of participants have similar skills.

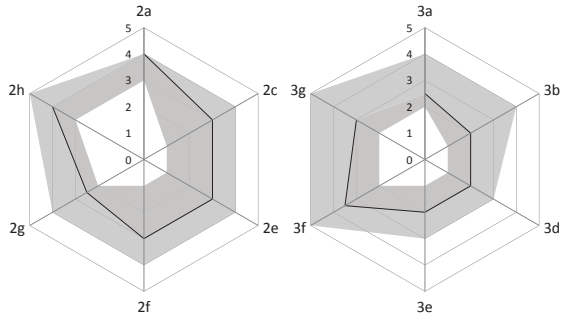

Figure 6. Pretest questions 2 and 3 (also see Table I

Figure 7 shows the answer to question 2b of Table I, namely the familiarity with various Integrated Development Environments. More than half of the participants reported to have experience with Visual Studio. Furthermore, all subjects have an Eclipse background, indicating that they are familiar with typical features that modern IDEs such as Visual Studio offer. Connected to the previous question, the radar chart of Figure 6 question 2c shows that only one participant indicated to have no experience with Visual Studio, while the median score is at 3 on a 5-point Likert scale.

Pretest question 2d (see Figure 8) shows the participants' programming language experience. All participants reported to have experience with Java, while half of the group also has experience with C#. Due to the similarities between the languages this is not seen as a problem for the experiment.

Questions 2e and 2f (Figure 6) show the experience with .NET languages and C# respectively. Two participants reported to have no experience, neither with .NET nor with C#, while all others reported to have experience, resulting in a median score of 3.

The next questions gauge the participants' experience with working with large scale projects (question 2g) and their understanding of the challenges that arise in (large-scale)

[7]The radar charts should be interpreted as follows: the minimum and maximum values of the respondents' answers are marked by the gray area, the median score is indicated by the black line

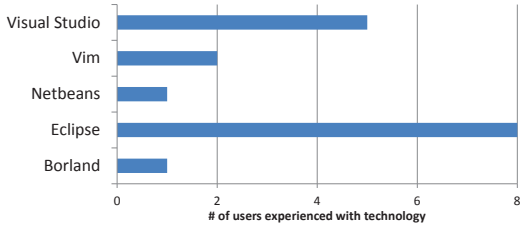

Figure 7. Pretest question 2b, IDE experience

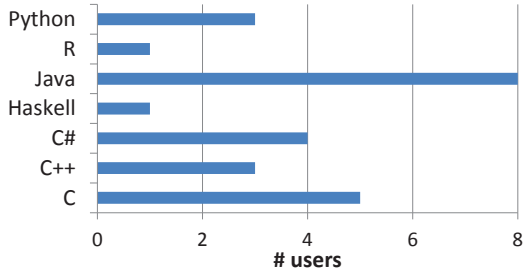

Figure 8. Pretest question 2d, programming language experience

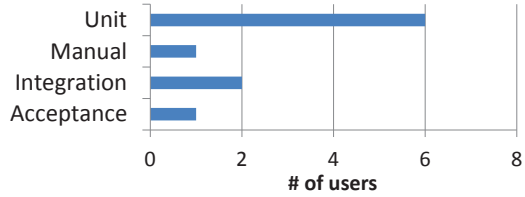

Figure 9. What kind of tests do the participants write?

software projects (question 2h). While the participants' experience with large scale projects is limited (median score of 2.5), they do indicate that they understand the challenges quite well (median score of 4).

None of the participants had knowledge of Jurassic, the subject system of our experiment (question 2i).

2) Testing experience: Only one of the participants considered himself an experienced tester (score of 4 on a 5-point scale), while most others considered themselves averagely experienced (median of 2.5, see question 3a in Figure 6). When asked whether they write tests for most of the code that they are writing, the participants indicated that they do not really write tests for the code they write, with the exception of one participant.

Subsequently, we asked what kind of tests the participants typically write, which results in the overview provided in Figure 9: unit tests are by far the most popular kind of test.

Question 3d of Figure 6 rates the statement "I believe that it is better to deliver fast than to spend more time on testing". With a median score of 2, the general trend is that the participants do see the added benefit of testing. Question 3e then asks to rate "I believe that one-man projects do not require automated tests". With again a median score of 2, most developers indicate that automated testing is also beneficial here.

The answers to the next two questions indicate that most participants agree that the volume of test code can surpass that of the production code (median score of 3.5 for question 3e). They also indicate that the effort spent on writing and maintaining testing code can be greater than that for production code (median score of 3, question 3f).

Finally, we asked them to indicate how much code coverage they considered "good". With answers ranging from 40% to 100% the average settled on 80%. A response of 100% means that the participant was not fully realistic because the return of investment of such a coverage is not

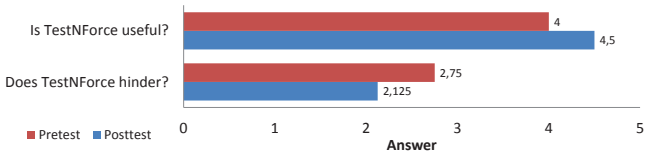

Figure 10. Comparison of opini[illegible]ent

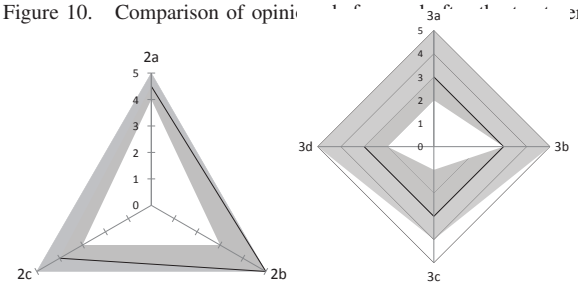

Figure 11. Posttest questions 2 and 3

justified [19, p.499]. We believe however, that the average of 80% as expressed by the participants is a realistic target.

C. Posttest

In the pretest, participants were given an abstract description of a tool like TestNForce (see Table I). Based on that description, we asked the participants whether they thought that TestNforce was (1) useful and (2) annoying or too intrusive. In the posttest we asked the same question again, but this time the participants had experience with TestNForce. Figure 10 presents the comparison of the average opinions before and after using TestNForce. What we see is that the participants expected TestNForce to be useful (score of 4), but after having used TestNForce, they were actually even more positive (score of 4.5). Along the same line, we asked whether they thought TestNForce would be annoying during development or too intrusive in the regular working habits. The response here decreases, meaning that before using TestNForce the participants expected worse. This is reflected in the score of 2.75 before their programming assignments with TestNForce, compared to a score of 2.125 after.

Figure 11, question 2a gives the participants' opinion on whether TestNForce helped them to complete the assignment. With a median score of 4.5, the participants give a clear indication that TestNForce was indeed helpful during the assignments. Question 2b asks whether the participants thought that TestNForce improves the identification of the tests that cover a particular method. With 6 participants providing the maximum score of 5 for this question, it seems that the participants are convinced that TestNForce does indeed help them with this task. Finally, we also asked the participants whether TestNForce helps them to become more confident when changing unknown code (question 2c). With a median score of 4 and all answers in the range 3 to 5, there is again an indication that TestNForce helps during maintenance.

The subquestions of question 3 mainly deal with how the participants experienced the check-in policy of TestNForce. In particular, question 2a of Figure 11 asks whether the participants saw the benefits of having such a check-in policy. Their responses give a somewhat mixed image, with scores ranging from 2 to 5 and a median score of 3. The rather average rating for the check-in policy might be explained by the response to question 3c, which gauges whether the check-in policy is too restrictive. In fact, many of the respondents do think it is too restrictive with a median score of 3 and the scores ranging from 1 to 4. Question 3d states that it should not be possible to bypass the check-in policy. The answers to this question range from 2 to 5 with a median of 3, indicating that opinions are differing greatly in this respect. This diversity in answers can be partly explained by the answer to the previous question, where participants indicated that they found the check-in policy to be too restrictive. On the other hand, people with a high score might realize the importance of having self-contained commits [15], or commits that contain not only the production code, but also well-covering tests to ensure the quality of that production code.

The next set of questions query the participants for missing features. Table II shows the average responses of the participants to the list of features that we consider as future work. The most desirable feature is the incremental update of the index. This feature would allow TestNForce to update the index fast by replacing only the records that are affected by code/test changes, instead of redoing the complete analysis that we presented in Section II-B. An important missing feature that was not on our standard list, but was mentioned by all participants is the lack of navigation from the covering tests window (see Figure 2) and the actual test code. In fact, during the short debriefing discussion that we organized after the posttest, many of the respondents also mentioned that the lack of this feature impacted the usability of TestNForce. However, the participants also mentioned that it was not a blocking issue, because it could be circumvented by searching the code and using the search results windows for navigation.

Feature	Average score
Support for other programming languages	4
Possibility to exclude certain parts of the project from checking	3
Incremental update of index	5
Static code analysis	3.5
Integration with the test platform in Visual Studio	4

Table II
MISSING FEATURES IN TESTNFORCE

D. Evaluation of the experiment

The final questions of the posttest dealt with how the participants perceived the experiment. In particular, we asked whether the participants found the assignment too difficult or whether they needed more time. For both questions, the median score was 2, indicating that the assignment was not

perceived as too difficult and that the time allocated for the assignment was satisfactory.

V. Discussion

This section will first relate the results of the experiment to the research questions that we have presented in the introduction. Afterwards, we will touch upon a number of threats to validity.

A. Discussion of the experiment

RQ1: Can such a tool be built with acceptable performance?: We have designed TestNForce as a Visual Studio 2010 plug-in. We have tried to reuse many standard tools and/or libraries like `vsinst` to instrument, `mstest` to execute individual tests and the C# parser from the NRefactory library to check for changes. Given that we were able to reuse many existing technologies, we are confident that a tool similar to TestNForce can also be created for other platforms, e.g., Eclipse. During development work, TestNForce is lightweight and does not cause a noticeable performance impact for the developer, however, building the test mapping index takes quite some time. More specifically, for the Jurassic case study with its 344 test cases this took 28 minutes. Usability-wise, this is a serious deterrent. It comes as no surprise then that *all* 8 participants of the experiment indicate incremental updates to the index as the most wanted feature for future versions. While non-trivial, incrementally building and updating the index can likely be combined with continuous testing, as proposed by Saff and Ernst [20].

RQ2: Is the tool considered useful by developers?: The pretest-posttest experiment setup lets us compare the expectations based on an abstract description of the tool and the experience of the developers with the actual tool. While the participants were already quite positive after reading about the tool (average score of 4 on a 5-point Likert scale), they were even more positive after using the tool during the assignment (average score of 4.5). This evolution is depicted in Figure 10.

RQ3: Do the developers experience the tool as a hindrance during development?: Related to the previous question, the participants have lowered their score for hindrance in the posttest compared to the pretest. Figure 10 shows that the average score for hindrance was lowered from 2.75 towards 2.125. However, it should be noted that the check-in policy did cause more controversy. Question 3 of Figure 11, which deals with the user experience of TestNForce's check-in policy shows greatly differing opinions. A number of participants found the check-in policy to be too restrictive and shared that they would like to configure the check-in policy so that it could be circumvented in some cases. So, while the normal usage of TestNforce is not considered as a hindrance, the check-in policy needs further attention. The mixed experience of the check-in policy might be related to the fact that (1) the check-in policy is conservative, in the sense that it will likely cause a number of false positives, i.e., cases were the check-in policy raises an alarm, while the tests should actually not be altered, and (2) the developer is convinced that the alterations to the production code should not be backed up by changes to the test code.

B. Threats to validity

1) Internal validity: The first problem with a one-group pretest-posttest experiment is the effect of history ("the specific events occurring between the first and second measurement in addition to the experimental variable"). In order to minimize the risk of this threat, the experiment was conducted without breaks between observation.

It might be that the participants were confronted with the effects of testing, meaning that the participants knew what they were tested for during the posttest. We tried to mitigate this effect by telling the participants in advance that only honest answers were of use for our experiment.

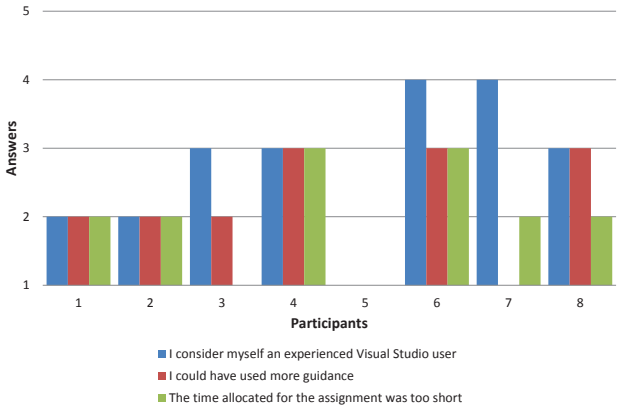

Figure 12. Comparing programming experience with assignment experience

We were concerned that participants with Visual Studio and C# experience had an advantage. However, from the results, we saw no correlation between participants who reported to be inexperienced with Visual Studio or C# found and assignment difficulty or time-pressure (see Figure 12). Furthermore, we do not base any of our conclusions on the speed by which the assignment was performed.

2) External validity: The participants to our experiment are students recruited at the Delft University of Technology. They might not be representative of experienced developers. While we agree that they do not have the same level of experience that professional developers might have, they have a very diverse background, as evidenced by Figure 5 which shows their diverse educational background.

The tasks that the participants had to perform during the assignment might not be representative of real-world programming tasks. We tried to mitigate this concern by trying do devise realistic programming tasks. On the other hand, the opinions of the usefulness of TestNForce are likely not only based on the participants' actual experience

during the programming tasks of the experiment, but also on previous development experience.

None of the participants had experience with the subject system, which is likely to be different in an industrial setting, were code ownership is more likely. In this sense, when developers are thoroughly familiar with the code, the added value for a tool like TestNForce might be less obvious. In the future, we will carry out an longitudinal study with industrial developers.

3) Reliability validity: The TestNForce toolchain might contain faults which explain the results of the user study. As a countermeasure, we thoroughly tested the tool and relied on a pilot study to iron out the last problems.

VI. Related work

Saff and Ernst investigate whether continuous testing, i.e., testing that happens in the background of development activities, helps developers [20]. In particular, they set up a controlled experiment to determine whether continuous testing results in a higher change of successful completion of programming tasks and decreases the time spent on programming tasks. From their experiment, they conclude that continuous testing does indeed increase the chance of successfully completing tasks, but there was no significant correlation with the time worked on tasks.

Regression testing is the process of validating modified software to detect whether new errors have been introduced into previously tested code and to provide confidence that modifications are correct. This step typically happens after unit testing, but sometimes both levels of testing are blurred. As regression testing is an expensive process, researchers have been working towards regression test selection techniques as a way to reduce some of this expense. Rothermel and Harrold discusses safe regression testing techniques in [21]. In essence, regression test selection techniques try to find those tests that are directly responsible for testing the changed parts of a program and subsequently only run these tests, which roughly corresponds with the second scenario that we envisioned for TestNForce as well. However, TestNForce is clearly geared towards developer testing, while regression test selection techniques are not.

Zaidman et al. try to determine whether production code and test code co-evolves [22]. In order to establish traceability between the unit test and the method under test, they use naming conventions. For the two case studies that they investigated, these naming conventions were upheld, but they also acknowledge that this is not always the case. In their study they notice that co-evolution between production code and test code is sometimes not optimal. This is reflected by the fact that the test source code sometimes does not compile for several versions or by dropping levels of code coverage.

Van Rompaey and Demeyer use static call graphs to determine the methods under test for a particular test case [23]. The results indicate that they only obtained 25% precision and they recommend to look at naming conventions or dynamic analysis to establish traceability links between production and test code.

Galli et al. have developed a tool to order broken unit tests [16]. It is their aim to create a hierarchical relation between broken unit tests in order to steer and optimize debugging. Technically, their approach is similar in the sense that they also instrument the tests before running in order to establish links between production and test code. Conceptually, however, the goals of their approach and our approach are quite different: ordering broken unit tests versus assisting developers during maintenance of production and test code.

Recently Microsoft introduced the "Test Impact Analysis" feature in Visual Studio 2010[8]. This tool is very similar in nature to TestNForce, but, to the best of our knowledge, has not been described in literature nor has not been subjected to a user study.

VII. Conclusion

In this paper, we have introduced TestNForce, a Visual Studio plug-in that helps software developers to better maintain the unit and integration tests that accompany production code. TestNForce allows developers to query which tests cover a particular unit of code (e.g., a method), it allows developers to remember which tests should be run after performing a modification and it warns developers before committing in case the tests that cover modified production code have not been updated.

In order to assess the usefulness of TestNForce, we have conducted a pretest-posttest experiment involving eight developers, in which we presented the developers with a questionnaire before and after a programming assignment in an environment in which they were faced with TestNForce.

Our results show that developers actually see the added value of developer test management tools, but have strict requirements with respect to their usability. We will now go over the research questions that we have stated in Section I:

- RQ1 *Can such a tool be built with acceptable performance?* We have designed TestNForce as a Visual Studio 2010 plug-in, reusing as much of the standard infrastructure as we could. During development work TestNForce does not cause a noticeable performance impact, however, building the test mapping index takes a long time: 28 minutes for the 344 test cases of our case study.
- RQ2 *Is the tool considered useful by developers?* With an average score of 4.5 on a 5-point Likert scale, the participants indicated to find TestNForce very useful for steering their test and test maintenance activities.
- RQ3 *Do the developers experience the tool as a hindrance during development?* The average score for

[8] http://msdn.microsoft.com/en-us/library/ff576128.aspx, last visited Oct. 14, 2011.

hindrance that the participants gave was 2.125, which was down from the expectation that the participants formulated in the pretest. However, it should be noted that the check-in policy did cause controversy and not all participants were happy with how this feature worked. In particular, they found the policy too restrictive and participants indicated that they would prefer to be able to configure the check-in policy.

Contributions. Over the course of this research, we have made the following contributions:

- The TestNForce Visual Studio plug-in, which enables to manage tests during software maintenance.
- A user study with eight developers to assess the usefulness and hindrance during use TestNForce.

Future work. We now list some of the important avenues for future work which both address usability and further research: (1) the possibility to navigate from the tests in the results window (see Figure 2) directly to the actual test code, (2) the incremental build-up of the test mapping, thus avoiding long waiting times, and (3) a more extensive evaluation using a controlled experiment that would give us insight into the time gain of using TestNForce (similar to, e.g., [24], [25], [20]).

ACKNOWLEDGEMENTS

We want to thank all the participants to our experiment. Also thanks to Cuiting Chen and Tiago Espinha for providing feedback on earlier versions of this paper. Part of this research was funded by the Center for Dependable ICT (CeDICT), an initiative of NIRICT, the Netherlands Institute for Research on ICT. Other funding came from the RAAK-PRO project EQuA (Early Quality Assurance in Software Production) of the Stichting Innovatie Alliantie.

REFERENCES

[1] V. Hurdugaci, "Aiding software developers to test with TestNForce," Master's thesis, Delft University of Technology, 2011.

[2] M. Lehman, "On understanding laws, evolution and conservation in the large program life cycle," *Journal of Systems and Software*, vol. 1, no. 3, pp. 213–221, 1980.

[3] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," in *Proc. of the International Workshop on Principles of Software Evolution (IWPSE)*. IEEE CS, 2005, pp. 13–22.

[4] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[5] G. Tassey, "Economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology (NIST), Planning Report 02-3, May 2002.

[6] A. Zaidman, M. Pinzger, and A. van Deursen, "Software evolution," in *Encyclopedia of Software Engineering*, P. A. Laplante, Ed. Taylor & Francis, 2010, pp. 1127–1137.

[7] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empir. Softw. Eng.*, vol. 16, no. 3, pp. 325–364, 2011.

[8] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, "The interplay between software testing and software evolution," in *Software Evolution*, T. Mens and S. Demeyer, Eds. Springer, 2008, pp. 173–202.

[9] S. Elbaum, D. Gable, and G. Rothermel, "The impact of software evolution on code coverage information," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE CS, 2001, pp. 170–179.

[10] M. Skoglund and P. Runeson, "A case study on regression test suite maintenance in system evolution," in *Proc. Int'l Conf. on Softw. Maintenance (ICSM)*. IEEE CS, 2004, pp. 438–442.

[11] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 23, pp. 22–29, 2006.

[12] M. Grindal, J. Offutt, and J. Mellin, "On the testing maturity of software producing organizations," in *Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART)*. IEEE CS, 2006, pp. 171–180.

[13] E. Engström and P. Runeson, "A qualitative survey of regression testing practices," in *Product-Focused Software Process Improvement*, ser. LNCS, M. Ali Babar, M. Vierimaa, and M. Oivo, Eds. Springer, 2010, vol. 6156, pp. 3–16.

[14] Y. Chen, D. Rosenblum, and K. Vo, "Testtube: A system for selective regression testing," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE CS, 1994, pp. 211–220.

[15] F. Mulder and A. Zaidman, "Identifying cross-cutting concerns using software repository mining," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*. ACM, 2010, pp. 23–32.

[16] M. Galli, M. Lanza, O. Nierstrasz, and R. Wuyts, "Ordering broken unit tests for focused debugging," in *Int'l Conf. Softw. Maintenance (ICSM)*. IEEE, 2004, pp. 114–123.

[17] D. Campbell, J. Stanley, and N. Gage, *Experimental and quasi-experimental designs for research*. Rand McNally, 1963.

[18] E. Babbie, *The practice of social research*. Wadsworth Belmont, 2007, 11th edition.

[19] C. Jones and O. Bonsignour, *The Economics of Software Quality*. Addison-Wesley, 2012.

[20] D. Saff and M. D. Ernst, "An experimental evaluation of continuous testing during development," in *Proceedings of the SIGSOFT international symposium on Software testing and analysis (ISSTA)*. ACM, 2004, pp. 76–85.

[21] G. Rothermel and M. Harrold, "Empirical studies of a safe regression test selection technique," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 401–419, 1998.

[22] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, "Mining software repositories to study co-evolution of production & test code," in *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*. IEEE CS, 2008, pp. 220–229.

[23] B. Van Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE CS, 2009, pp. 209–218.

[24] B. Cornelissen, A. Zaidman, A. van Deursen, and B. V. Rompaey, "Trace visualization for program comprehension: A controlled experiment," in *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2009, pp. 100–109.

[25] B. Cornelissen, A. Zaidman, and A. van Deursen, "A controlled experiment for program comprehension through trace visualization," *IEEE Trans. Software Eng.*, vol. 37, no. 3, pp. 341–355, 2011.

TUD-SERG-2012-002
ISSN 1872-5392