Aiding Software Developers to Test with TestNForce

Victor Hurdugaci

Aiding Software Developers to Test with TestNForce

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Victor Hurdugaci
born in Braşov, Romania

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Aiding Software Developers to Test with TestNForce

Author: Victor Hurdugaci
Student id: 4032381
Email: contact@victorhurdugaci.com

Abstract

Regression testing is an expensive process because, most of times, all the available test cases are executed. Many techniques of test selection/filtering have been researched and implemented, each having its own strong and weak points. This paper introduces a tool that helps developers and testers to identify the tests that need to be executed after a code change, in order to check for regressions. The implementation is based on dynamic code analysis and the purpose of the tool is to eliminate the time spent on testing using inappropriate test cases (tests that bring no value in checking for regressions). The adequacy, usability and completeness of this tool have been evaluated through the meanings of a user study. During the study, a number of developers used the tool and expressed their opinion about it through questionnaires.

Thesis Committee:

Chair:	Prof. Dr. Arie van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Andy Zaidman, Faculty EEMCS, TU Delft
Committee Member:	Dr. Jan Hidders, Faculty EEMCS, TU Delft

To my parents,
for all the trust and support they gave me
since I left Romania

Preface

One year and one day passed since I started my thesis project. When I look back at what has happened in the last year, I am very satisfied of what I have learned and about the achieved results. I am the one getting the credits for this thesis but the project would not have been possible without the help of many other.

First of all I would like to thank my thesis coordinator, Andy Zaidman. Without him, it would have been impossible for me to finish the master program on time and still do the 1 year internship in Microsoft. He understood my desire of doing both in the same time and helped me overcome all the issues caused by the fact that I was in another country.

Special thanks go to the participants of the user studies (user experience survey, pilot run and user study). Without them, it would have been impossible to evaluate TestNForce. Also, my colleagues at Microsoft provided valuable feedback for the project, both in the beginning, by suggesting must-have features, and at the end, suggesting features for the next versions. I am very grateful to them. I could not have delivered a correct paper without the help of Lavinia Tănase, Kruna Matijevic and Daniel Manesku. I am very grateful to them too for reviewing parts of my thesis and suggesting improvements.

Finally, I want to thank to my family for all the trust and support they offered me. It would have been very hard to finish this project without their moral support.

Victor Hurdugaci
Delft, the Netherlands
July 3, 2011

Contents

List of Figures

Chapter 1

Introduction

Errare humanum est (lat. "To err is human"). Companies test software not because they like to do it, but because they have to do it [15]. Software developers are human and humans make mistakes. That is why software has to be tested before giving it to end users. Software can be tested manually, but a more common practice, that increases the return of investment and decreases the testing time, is to automate tests[20]. Automated tests can run without any intervention and, if done right, the effort put in developing and maintaining them is less than executing each of them manually.

Especially in the case of large scale software projects, regression testing is a tremendous and time consuming job[18]. On one hand, testing is complicated because of the size of the project. On the other hand, it is complicated because running the tests can take many hours, even days[20], so one cannot validate the system instantly or in reasonable time.

It is well known [5] that except for the rare event of major rewriting, changes affect only a small part of the system. That part of the system is covered by only a subset of the available test cases (figure 1.1). Identifying all the tests (without positive or false negatives) can be troublesome especially in the case when apparently unrelated components interact.

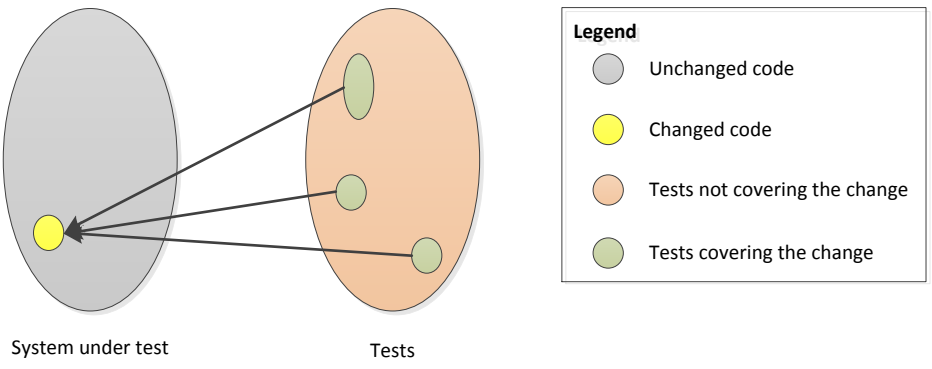


Figure 1.1: The problem with testing changes in large scale systems

1.1 Thesis project

Based on the author's experience at Microsoft, we can state that identifying the tests that can check for regressions after code changes is difficult. This fact was also confirmed by Chen who states that "computing such dependency [between tests and code under test] information requires sophisticated analyses of both the source code and the execution behavior of the test units[5]". The identification process is error prone because not everyone is aware of all the links in a system - components that are apparently independent can actually communicate (indirectly) - and because, in many cases, test descriptions do not match the actions of the test - tests cover more or less than what is told in description.

Zaidman et. al[21] found that tests and code under test do not co-evolve, as they should, all the time. Even though there is no clear evidence, we believe that sometimes this happens because maintaining the tests is too expensive. If developers/testers would have a tool to identify the tests that need to updated, making the process cheaper and faster, there is a high chance that co-evolution will happen as expected (tests evolve in the same time as the code they cover).

In order to help developers and testers identify the tests that cover specific parts of the code, a tool, that will automate this process, will be created. The thesis project is represented by this tool and its evaluation.

1.1.1 Research Questions

Central to this research projects stands the development of the prototype tool. If the implementation is possible, then the tool needs to be evaluated in order to decide if its purpose is satisfied. Based on this conception, a number of research questions will be answered throughout the paper:

- *Question 1*. Can such a tool be implemented using the currently available tools?
- *Question 2*. Is the tool usable from a performance point of view?
- *Question 3*. Is the tool useful for developers and testers?
- *Question 4*. Is the tool offering a good user experience?

1.1.2 Goals

Based on the research questions stated in section 1.1.1, a number of goals can be identified.

1. *Answer the research question*. The most obvious goal is to answer the questions stated above. However, this is not possible without achieving a few other goals.
2. *Implement a tool that identifies the tests which cover a specific part of the code*. Implementing the tool is the foundation of the project. Without the tool, it is impossible to answer many of the research questions.

3. *Evaluate the tool*. Once implemented, the tool has to be evaluated by individuals outside the development group. This will allow us to obtain objective answers and decide if the tool reaches our expectation level.

Since this tool enforces the test process and increases the developers/testers' capacity of identifying tests we shall call it TestNForce (pronounced "test enforce"). In section 5.2, one can read a story behind the name and what happened before the name was found.

1.2 The vision

... or what is TestNForce supposed to be? TestNForce is supposed to be tool that allows the user to see which tests cover the (changed) code. Full stop. Satisfying this only high level requirement, will prevent TestNForce from being bloated with unnecessary functionality and will lead to a tool that does one thing but it does it well.

Following the previous requirement, the user will need to integrate the test identification process in his own development flow. For achieving this, a number of interfaces will be provided, simplifying the integration.

1.3 Use cases

Based on the singular requirement above, the use cases for TestNForce will be very simple. It can be noticed that the user can request the tests either for "code" (a specific piece of code) or for the changed code. Therefore, the user can act in two ways:

1. Get tests for the changed code
2. Get tests for a particular method

By looking at the previously mentioned actions, it can be determined that the first action will not be possible without knowing which code changed. That is why a third action can be identified: *Detect changed methods*. It must be noticed that this action is not directly invoked by the user but rather, it is used by *Get tests for the changed code* and it does not make sense, in isolation, without the action that uses it. Both user actions are not possible if the relation between the tests and code under test is unknown. In order to have the relation, an index must be built. This leads to the fourth and final action: *Build index*. This action is not a user action but a consequence of the required user actions. Figure 1.2 shows the identified actions through the meanings of an UML use case diagram.

1.4 Requirements analysis

Before being able to proceed any further, the use case diagram must be translated to a list of specifications. These specifications must be satisfied by the system that will be implemented.

From the use case presented before, a number of obvious requirements can be extracted:

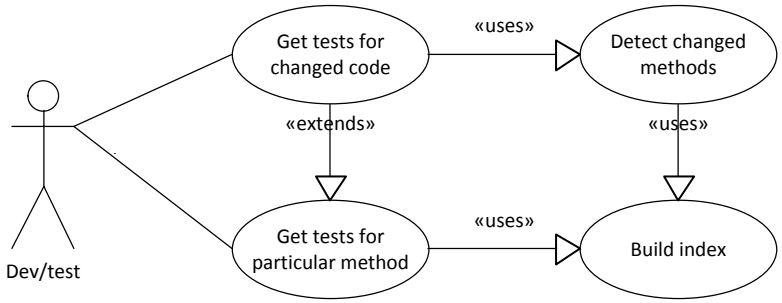


Figure 1.2: Use cases of TestNForce

1. *Get the list of tests that cover a specific method*. TestNForce must show to the user the tests that cover a specified method.

2. *Specify for which method the results should be displayed*. From the previous requirement, a new one can be deduced. The user must be able to invoke, somehow, the actions on a specific method.

3. *Get the list of tests that cover the changed code*. TestNForce must be able to show the tests that cover the code that changed since a specific point in time, generic called, last change.

4. *Detect changed code*. Again, from the previous requirement, we can extract another requirement. TestNForce must be able to detect the code that changed.

However, just a set of functional requirements cannot make a good application. A number of non-functional specification will guide the development process towards a successful and useful application:

1. *Simplicity*. TestNForce is supposed to be as simple as possible. Only the decisions that cannot be done without the user's intervention should be exposed, all the other should be handled by the system.

2. *Integration*. TestNForce must integrate in the users' current workflow and should not require any major changes in the process.

3. *Transparency*. Once integrated, TestNForce must not disturb the user and, as much as possible, should make its presence noticed only when the user asks it.

4. *Good performance*. The responses should be displayed in a reasonable amount of time from the moment the user invoked an action. However, it is not expected that the application will perform fast if there are good reasons why this is not possible (ie. a large quantity of data is analyzed)

5. *Accuracy*. TestNForce must provide 100% accuracy. This requirement is not extracted from the use cases but was decided by us.

Based on the experience of the development team and because of the limited available time, the following technical decisions have been taken:

- The implementation will be done in C# because of the developers' experience
- For simplicity, the analyzed code will be also C#
- Because of the IDE integration, TestNForce will be integrated with Visual Studio
- Visual Studio provides a test platform (through `mstest`) that is semi-extensible. If extension is possible, then this is the first choice.

1.5 From requirements to implementation decisions

The last non-functional requirement enumerated in the previous section is one of the most important aspects of TestNForce. In order to obtain 100% accuracy, all the analyzed code must be understood and there should be no heuristics involved. An interesting approach was presented by Galli in [10], where they use dynamic code analysis to establish the links between tests and code under test. A similar approaches, but with different goals, was adopted by Rothermel[19] in the attempt of prioritizing test caess.

Dynamic analysis is the analysis of the properties of a running program[1]. If implemented correctly, it can give perfect results, but all these do not come for free. In order to dynamically analyze the code, it has to compile. The compilation is required because the binaries have to be executed.

A second disadvantage is the time required for analysis and the impossibility of predicting it. In the case of a static analysis one can, with some error, predict the execution time based on the number of lines of code. In dynamic analysis, this is not possible because the run time of each instruction and the number of times it runs cannot be predicted; if infinite loops are taken into account, it can be said, even though unrealistic, that the analysis can last forever.

1.6 Thesis structure

TestNForce is a software product. Its implementation and internals are detailed in chapter 2. The most interesting parts of the prototype are shown using diagrams, technical details and other appropriate means of representation.

After understanding the vision and the technical details behind TestNForce, the reader can look in chapter 3 to see what others think about the tool. In chapter 3, a user study is presented in details. Both the setup and the results are analyzed and two research questions are answered.

Similar initiatives and similar research to TestNForce exists. Before jumping into conclusions, an overview of the scientific publications related to our research is outlined in chapter 4.

The final chapter of the thesis is devoted to conclusions. Except a summary of the answers to the research questions, chapter 5 tells three remarkable stories that happened during the development process and briefly presents the most important items from the next version's backlog.

Following the final chapter, a series of appendices contain the documents referenced in chapter 3. Appendix A.1 contains the questionnaire used for the user experiment survey. Appendix A.2 and A.3 show the pretest and the posttest, respectively. The assignment given to the participants of the user survey is shown in appendix A.4 while the results, in tabular form, of the pretest and posttest are printed in appendix A.5 and A.6, respectively.

Chapter 2

Development of TestNForce

TestNForce must help developers and testers to identify the tests that must be executed in order to validate changes to the existing codebase. It must provide 100% accurate results and it must not, under any circumstance, provide false negatives (miss tests that cover the code). The tool must provide a clear advantage over executing the entire test suite, but its presence should be as transparent as possible.

2.1 A bird's eye view of TestNForce

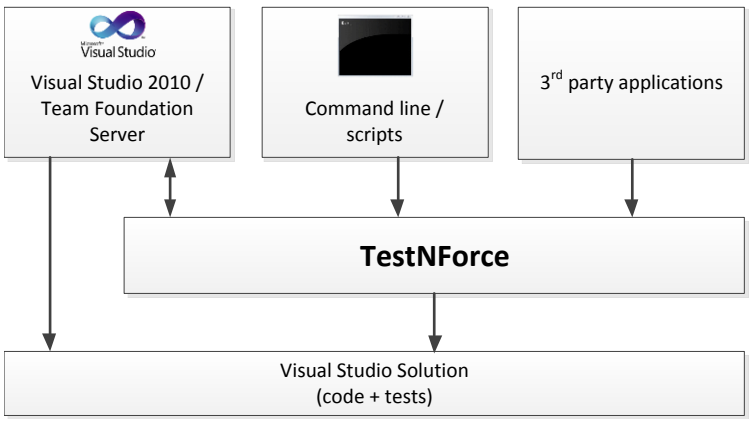


Figure 2.1: Bird's eye view

From an architectural point of view, TestNForce is designed to be extensible and easily integrateable with other applications. The prototype implementation offers a Visual Studio 2010 addin, which exposes all the features of TestNForce and a console application, which allows just a subset of the actions supported by TestNForce to be invoked.

As can be seen in figure 2.1, TestNForce will integrate between the application that uses it and the code. Any component that will require information about the test mapping

will use TestNForce to obtain it. However, when extra information is required, the Visual Studio solution can be accessed directly, because TestNForce is not blocking or restricting other workflows. TestNForce must be seen as a add-on that can be added/removed at any time and the only consequence of these actions is the possibility of using the test mapping feature.

2.2 Components

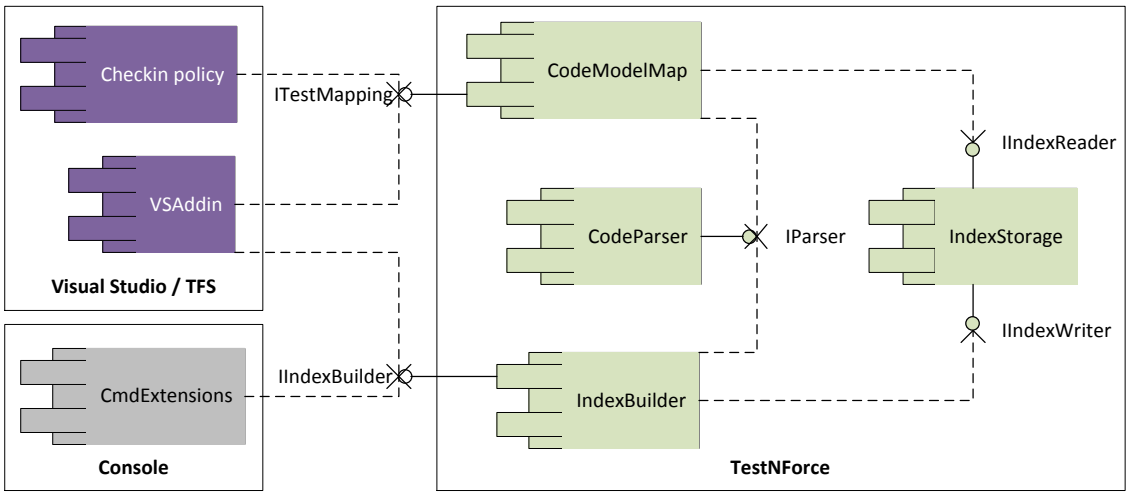


Figure 2.2: TestNForce's component diagram

TestNForce is composed of seven components. Each of them fulfills a specific role in the scenarios in which the application can be used. The components, shown in figure 2.2, are described below:

- *IndexStorage*. This component is part of the TestNForce core. It can be seen as a data abstraction layer[9] on top of the underlying storage service (the file system in this case). The main purpose is to provide an abstraction layer so that TestNForce is decoupled from the actual storage. This allows better testing, because the file system can be replaced by a mock and, in the same time, forbids TestNForce from accidentally changing files that do not belong to it.
- *Code parser*. The core of TestNForce was designed in such a way that is not interacting directly with external components. All the external calls need to go through intermediate, abstraction layers. The code parser is an abstraction over the code parsing library described in section 2.4. Similar to the case of IndexStorage, it simplifies testing and allows the actual implementation to be replaced by a new one(s) without changing the logic; such a design allows TestNForce to be adapted to other programming languages.

- *IndexBuilder*. It is responsible for creating the index that holds the mapping between tests and code under test. There is a clear separation between the component that creates the index and the one that reads it, because some applications will require just one of them, as can be seen in the section 1.3. This component performs the "7 step analysis" that is described in the next section.

- *CodeModelMap*. The index can be either read or writen to. The reading operations are handled by the CodeModelMap. This component provides APIs for getting information about the tests that cover a specific method. No external code should access the index directly (however, there is nothing that blocks the access).

- *VSAddin*. It is one of the components that is not part of the TestNForce core. The VSAddin allows Visual Studio 2010 to communicate with the TestNForce making the latter look as a part of the former. From a layered architecture perspective, the VSAddin is a presentation/service layer which provides the visual elements of TestNForce for VS and also allows VS to send messages back. Because from VS is possible to update/create the index and see the tests that need to be run, the VSAddin component depends on both the CodeModelMap and IndexBuilder.

- *Checkin Policy*. As the name suggests, this is the component responsible for the gated checkin for Team Foundation Server. This component depends only on the CodeModelMap because it will only use information about the tests that need to be executed and it will not alter the index in any way. The policy goes together with the VSAddin, but is an optional component. To be more precise, the addin and/or the checkin policy can be used independently, but both of them depend on Visual Studio. The checkin policy depends also on the Team Foundation Server SDK, but since version 10 of Visual Studio, this is a part of it.

- *CmdExtensions*. The command line extensions is a work-in-progress component that allows TestNForce to be integrated in scripts with the goal of enhancing automated builds with test enforcement features. It exposes commands for interrogating the index and identifying the tests that need to be run. In future versions it will allow index update operations.

2.3 7 Step analysis

In order to create an index that will allow TestNForce to identify fast what tests need to be run, a number of operations need to be performed. First of all, because of the dynamic code analysis requirement, the code has to run. However, in order for the code to run it needs to be compiled. Secondly, the code cannot just run, it needs to be run by the tests. In order to run the tests, they have to be identified first. Least, but not last, the execution of the tests needs to be tracked so that we can know what code was executed by a specific test.

All these constraints and the Visual Studio SDK lead to a seven step analysis process. The steps are listed below and each of them will be described in the rest of this section.

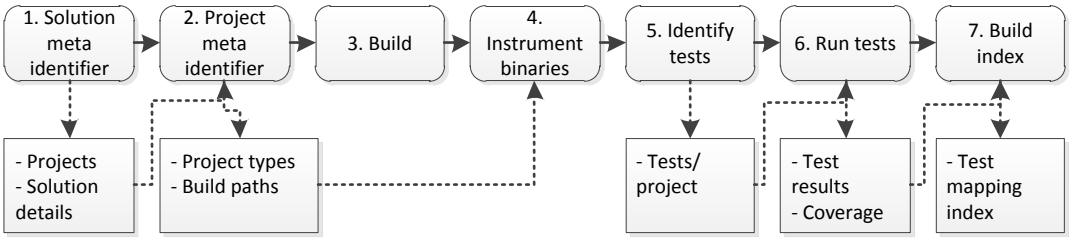


Figure 2.3: Full analysis overview

1. Solution meta identifier
2. Project meta identifier
3. Build
4. Instrumentation
5. Identify tests
6. Run tests
7. Build index

All the steps are implemented as a command pattern[11] and the data is passed between commands using data transfer objects (DTO)[1].

2.3.1 Solution meta identifier

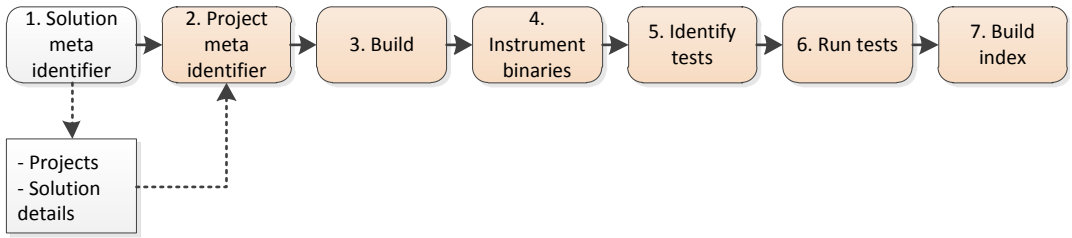


Figure 2.4: Step 1 of the analysis

The first step analyzes the Visual Studio solution and caches information about it. A DTO containing the path of the solution file and the list of projects available in it is created. The path of the solution will be used in the future steps when resolving relative paths. The

[1] http://msdn.microsoft.com/en-us/library/ms978717.aspx

list of projects is cached in the DTO, in order to prevent future calls to the SDK, because of performance reasons and because of the retry issue described in the next paragraph.

It will be noticed that many future step will take a similar approach of creating new objects that cache the information provided by the SDK. Such a design has two advantages (1) it reduces the number of retries needed caused by bug described here `http://msdn.microsoft.com/en-us/library/ms228772.aspx` (2) it allows the code to be testable.

If the commands would work directly with the VS SDK objects, then testing individual commands, in isolation, would be a difficult task. The DTOs have the purpose of creating an internal, intermediate representation of the data collected from the SDK. These DTOs are easy to mock and commands can be tested individually by supplying fake DTOs created by the test case.

The analysis will stop at the first step, if there are no projects in the solution. There is no need to continue because no test can exist outside a (test) project.

2.3.2 Project meta identifier

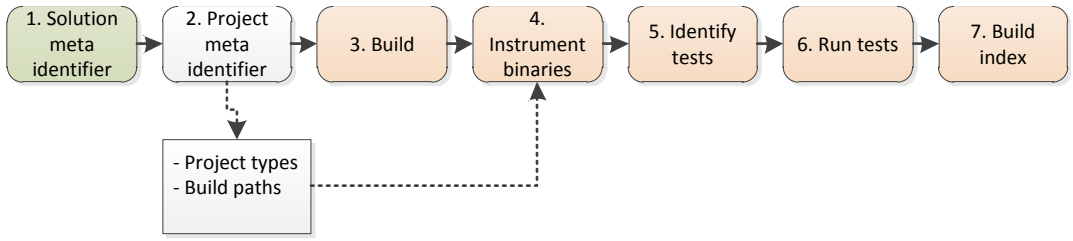


Figure 2.5: Step 2 of the analysis

Once the list of projects is created, the project meta(data) identifier command is invoked. This command will analyze each project in the solution, in order to decide if it can be used by TestNForce. If no valid (test) project is found, the analysis stops here. The projects that can be used by TestNForce are C# project and C# test projects.

Each Visual Studio project has a GUID which specifies its type. In the case of C# project, the project GUID is `FAE04EC0-301F-11D3-BF4B-00C04F79EFBC`. TestNForce first tries to identify all the C# projects. If at least one project is found, then a second analysis is performed. This second analysis has the purpose of identifying test project.

Any Visual Studio project can have subtypes. These subtypes are meant to allow the creation of new project types based on existing ones. A project with subtypes is called "flavored" and the flavors are specified by GUIDS in the `ProjectTypeGuids` section of the file.

One way of identifying a test project is to look for the test project flavor. However, this method is not reliable if the project was created manually or test references were added after the automatic creation. Visual Studio will run the tests in the project anyway. In order to mitigate future failures and/or incorrect results that could be caused by non-flavored

test projects, another identification method was developed. Any test class/method that contains `mstest` tests must be decorated with attributes defined in `Microsoft.VisualStudio.QualityTools.UnitTestFramework`. Therefore, in order to detect if a project is a test project, instead of looking for the test flavor, TestNForce looks for the referenced assemblies; if the previously mentioned assembly is referenced, then the project is marked as having tests.

Once the supported projects are identified and categorized, their output paths, output file names and internal names are stored in the DTOs. The output path is considered to be the path where binaries are placed after the build process using the default build configuration.

In order to continue the analysis, at least one test project must be found. Any invalid projects (the list could include, but is not limited to, Visual Basic.NET project, C++ projects, Azure projects) is simply ignored and not analyzed.

2.3.3 Build

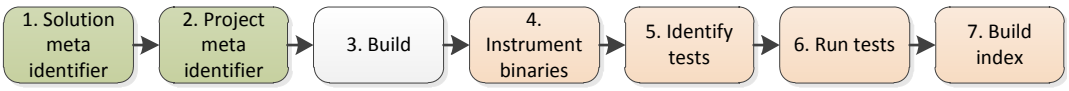


Figure 2.6: Step 3 of the analysis

The build process happens after the solution and projects are validated. It was decided to build only after validation because if the validation fails there is no reason to build. Also, some projects might have a long build time and it is faster to validate first, saving precious time.

The Visual Studio solution file (the files having ".sln" extension), holds the information about the default build configuration. This configuration can be changed from Visual Studio by changing the current build configuration or, from outside Visual Studio, by editing the sln file. The file is processed by `msbuild` which then processes each project file and builds it on the specified configuration.

TestNForce relies on the VS SDK for this operation. After the build is completed, the next step is invoked in case of success or the analysis stops in case of failure. TestNForce also relies on the user for running the build in the correct environment and does not try to solve any compilation problems.

2.3.4 Instrumentation

Figure 2.7: Step 4 of the analysis

As described by [3], instrumentation is "a mechanism that allows modules of programs to be completely rewritten at runtime". In the case of TestNForce the compiled binaries are instrumented so that their execution can be traced.

`vsinstr.exe` is a tool provided with the .NET Framework and facilitates the instrumentation of .NET binaries. It has a special mode for instrumenting with the purpose of code coverage.

TestNForce uses `vsinstr` on all the compiled binaries from the previous step. After the process completes, the binaries can be used by `vsperfmon` (see section 2.3.6) to trace the execution.

2.3.5 Identify tests

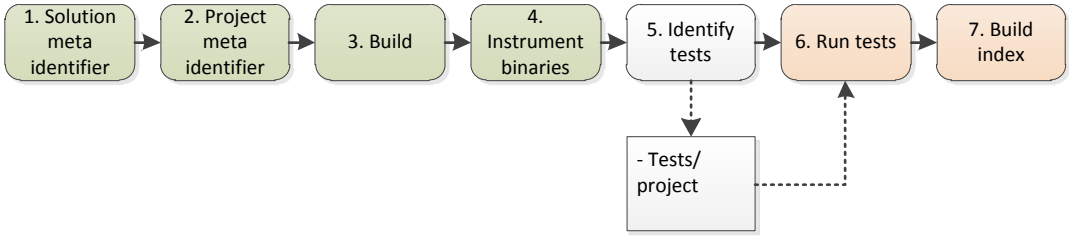


Figure 2.8: Step 5 of the analysis

After the binaries are instrumented, .NET Reflection is used to identify tests in each test assembly. The test assemblies were identified in step 2 (section 2.3.2).

From Reflection's point of view, an assembly is a collection of types, each type having methods. A test method, from `mstest`'s point of view, is a *public, instance* method with *no arguments* decorated with the `TestMethodAttribute`, attribute declared inside a *public* class decorated with the `TestClassAttribute`. Having these two definitions, allows TestNForce to identify the test methods using reflection. The pseudo code for identifying test methods is presented below (an exposed type/method is a public type/method):

```
for each test assembly TA in DTO
{
    for each type T exposed by TA
        which is decorated with TestClassAttribute
    {
        for each method M exposed by T
            which is decorated with TestMethodAttribute
        {
            Add M to the tests list
        }
    }
}
```

The identified method's details are stored as `tests` in the DTO corresponding to the project which created the assembly.

This fifth step of the analysis and the previous one could be performed in parallel. However, the decision was not to do so because both operations are fast (a few seconds) and handling multi thread execution, unless implemented with much care, would add a few new points of failure.

2.3.6 Run tests

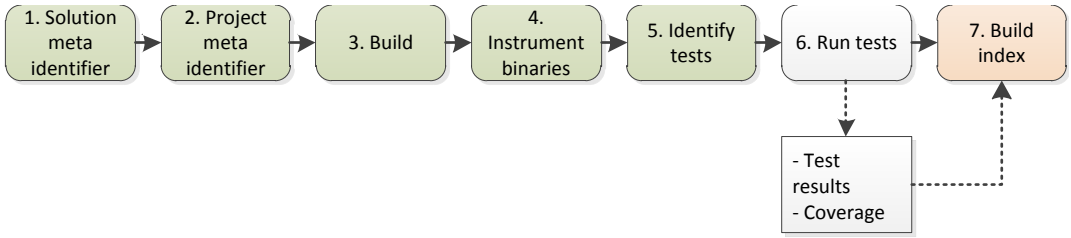


Figure 2.9: Step 6 of the analysis

The implementation of `mstest`, `vsinstr` and `perfmon` does not allow the generation of reports per test case. The only result provided is a general coverage report. TestNForce requires detailed coverage information about each executed test.

To the best of our knowledge, there are two ways of getting the required information from `mstest`:

1. *Write a .NET profiler.* Writing a custom profiler has many advantages one of them being that the code is instrumented "on the fly" making the instrumentation step not necessary. The .NET CLR (Common Language Runtime) exposes a set of interfaces that allow applications to subscribe to profiling events. One of these interfaces is `ICorProfilerCallback`[2] which provides callbacks for events like assembly loaded, method execution started, etc. However, writing a good profiler requires much experience with the .NET internals and just the profiler alone could make the subject of a thesis. Moreover, we do not have enough experience to guarantee that his implementation will work for any application. That's why, the second approach was addressed.

2. *Execute each test individually.* This approach is similar to what Galli et al did in [10]. Instead of intercepting the calls, the out-of-the-box `vsinstr` and `perfmon` can be used to track down the execution path of individual test cases. If only one report can be generated per test run, then if only one test is run the report will include exactly the information needed by TestNForce... for one test. In order to get the complete data about the execution, each test must be executed individually. Luckily, `mstest(.exe)`

[2] `http://msdn.microsoft.com/en-us/library/ms230818.aspx`

provides a series of command lines arguments that allow the execution of individual tests, as described next.

For each test case, the associated DTO stores information about it and about the coverage report that was generated for it.

How mstest, perfmon, perfcmd and vsinstr can give reports/test

As mentioned before, there is no out-of-the-box support for individual individual reports. This is mainly because `perfmon` cannot distinguish between the tests calling the executed code. However, running individual tests makes these reports possible.

A combination of arguments and processes is needed:

- `mstest` has the sole purpose of running the test case. It does nothing more that loading the test assembly and executing the test case. By default, it executes all the tests in the specified assembly(ies), but with specific arguments it can be forced to execute only certain test. In the case of TestNForce the `/test` argument, with the test name is passed. However, this argument works like `string.Contains`, matching any test that contains the value of the argument. In order to match only a single test the `/unique` argument is also added. With this two arguments and the path to test assembly, `mstest` is able to execute individual test cases.
- `perfmon` is the coverage collector. It will monitor any instrumented binaries who's process was started from the current process. This executable is launched just before a new test run starts and it is turned off after the test ends. Basically, while `mstest` runs some instrumented code, `perfmon` collects coverage data. The path to the generated report is stored in the DTO corresponding to the current test (if the test passed).
- `perfcmd` allows the manipulation of `perfmon` after it started. The latter one is blocked while running and it would be impossible to control it from itself. The argument `/shutdown` is passed to kill all instances of `perfmon`.
- `vsinstr` is taking part in this process only to instrument the binaries so that `perfmon` can recognize them.

The full process is described in pseudocode below:

```
run vsinstr.exe /coverage to instrument all binaries
for each test project P in projects DTO
{
    for each test T in P
    {
        create a file TMP for storing the report
        run perfmon.exe /coverage /output:TMP
        run mstest.exe /testcontainer:P /test:T /unique
        wait for mstest to exit
```

```
        run perfcmd.exe /shutdown

        if T passed then
            store path to TMP in the DTO for T
    }
}
```

This approach, even though it is slower than the custom profiler, has the advantage of having a more simple implementation. Also, it was more suitable for the limited research time allocated for the thesis. The results are identical.

2.3.7 Build index

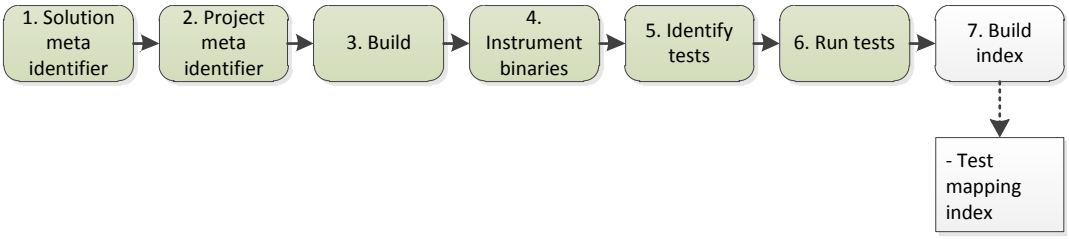


Figure 2.10: Step 7 of the analysis

The last step of the analysis process is the one that allows TestNForce to be TestNForce. Of course, all the predecessor steps are needed, but all those are done in order to allow the final step to succeed. During this step, the index that permanently stores the relation between tests and code under test is created.

Index structure

The index is, by a bird eye view, just an archive. It contains all the information about (test) methods and the relation between test methods and code under test. This archive always resides next to the corresponding solution file and has same name as the solution just a different extension, `tnf`.

Inside the archive two types of files exist:

- *Method description files*. This kind of files, hold the structure defined in section 2.4. For each indexed method, an associated file having the `mtddesc` description exists. The files are named with the method signature that they describe. An example of such name is `AppWithTests.Program.Add(System.Int32,System.Int32).mtddesc`. Inside the file, the method structure is stored in plain text. The reason behind naming the files with the full signature rather that just the method name is the polymorphism support of C#. If arguments would be ignored, then it would be impossible to distinguish between methods with the same name but different arguments. Also, the

full qualified name (including the namespace) is stored because an application can contains methods with the same name and, optionally arguments, in different namespaces.

- *Test mapping file*. The index mapping keeps a many-to-many relation 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. There is just one index file in the `tnf` file and its content is XML. 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:

```
<Root>
  <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>
</Root>
```

The name of the methods is identical with the name of description files corresponding to that particular method.

2.4 Code parsing and code comparison

TestNForce needs to compare snippets of code in order to determine if anything changed. Comparing two snippets of code is not a trivial task. Depending on the needs, the comparison algorithm can be more or less smarter than a simple string comparison.

The initial comparison algorithm was just checking the strings, bit by bit, ignoring any leading, trailing white spaces and empty. So, the following two snippets were equivalent because the formatting, in the second example, has only the ignored elements mentioned before.

```
void MyMethod()
{
int a=0;
a++;
Console.WriteLine(a);
}

//Equivalent to
```

```
void MyMethod()
       {
int a=0;
a++;

Console.WriteLine(a);
       }
```

However, such an approach had the disadvantage of not being able to deal with constructs that do not affect the functionality of the code, like comments. The following two snippets, even though have the same output, will be identified as being different, because of the comment:

```
void MyMethod()
{
int a=0;
a++;
Console.WriteLine(a);
}

//Not equivalent to:

void MyMethod()
{
  //A comment
int a=0;
a++;
Console.WriteLine(a);
}
```

In order to improve the accuracy of the detection, it was decided to parse the C# code and perform a comparison based on its semantic, rather than syntax. Implementing a C# parser would be a tremendous, error-prone and even useless job when there are many free available. Therefore, it was decided to use NRefactory[3] as it provides a good enough parser, sufficient for the needs of TestNForce. The structure will contain only information about the bits that will be executed in the compiled binary, ignoring formatting and comments. The picture below shows the semantical representation of a function that performs the addition of two integers:

```
static int Add(int a, int b){
//Comment
```

[3]http://wiki.sharpdevelop.net/NRefactory.ashx

```
  return a + b;
}

//Semantical representation:

[MethodDeclaration
 Body=[BlockStatement: Children={
  [ReturnStatement
   Expression=[BinaryOperatorExpression
    Left=[IdentifierExpression Identifier=a TypeArguments={}]
    Op=Add
    Right=[IdentifierExpression Identifier=b TypeArguments={}]
   ]
  ]
 }]
 HandlesClause={} Templates={}  IsExtensionMethod=False
 InterfaceImplementations={}  TypeReference=System.Int32
 Name=Add
 Parameters={
  [ParameterDeclarationExpression Attributes={}
   ParameterName=a TypeReference=System.Int32
   ParamModifier=In DefaultValue=[NullExpression]],
  [ParameterDeclarationExpression Attributes={}
   ParameterName=b TypeReference=System.Int32
   ParamModifier=In DefaultValue=[NullExpression]]
 }
 Attributes={} Modifier=Static
]
```

However, the semantical representation has a number of drawbacks. First of all, it uses more storage space; this is not a big issue for today's computers because the storage space is becoming cheaper and cheaper. Secondly, it cannot cope with renamed variables, renamed functions, reordered arguments or lines of code, changed types and, finally, preprocessor directives. Handling all those would require either implementing a hook to the C# compiler, which is currently not possible but is planned in future releases of the language or implementing a compiler which is out of the scope of TestNForce. It was decided that having a semantical representation is a good enough improvement from the primitive string comparison.

2.5 What tests do I need to run?

What would be the purpose of the index if it would not be used? Once the analysis is complete and the index is created, TestNForce can use it to display information about the tests that need to be run. TestNForce can provide the list of tests that covers a specific

method. Applications that use it have the responsibility of deciding the method(s) for which they want the tests list.

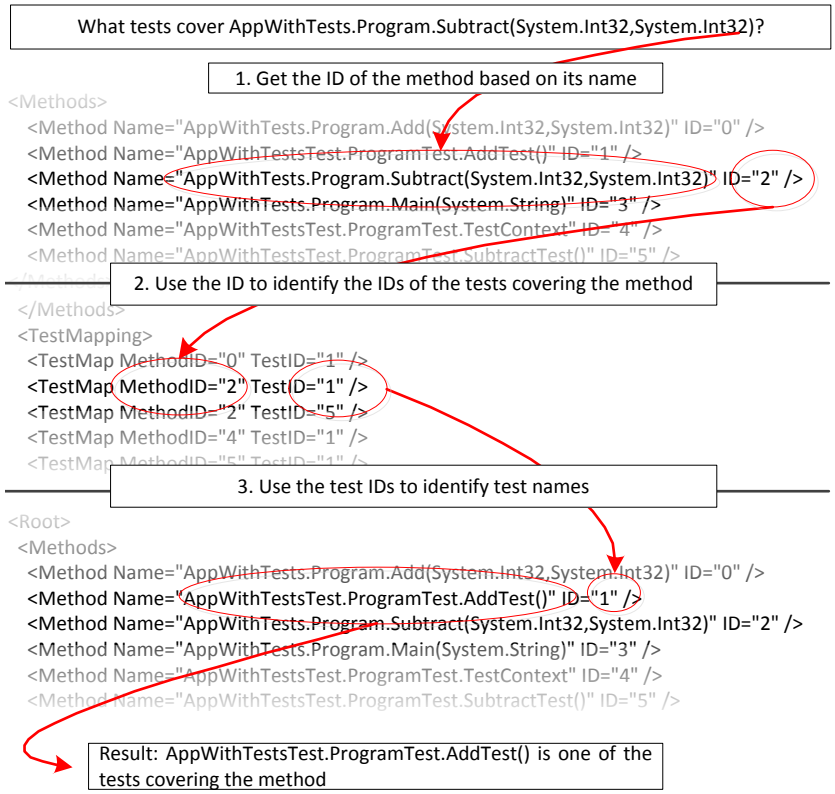


Figure 2.11: The process of mapping tests and code under test

The identification from index is a simple process similar to a join operation in a database. An abstract view of this process can be phrased like “for each method that I am interested in, look for the tests that cover it and give me an aggregated result”. Because of the index structure, the query is split in two sub queries:

- *Identify if the method is changed.* Once a request is done, the first step is looking for the method in the indexed methods list. If the method is not there, the operation fail. Otherwise, the semantical representation of the requested method is created and compared with the indexed one. In case of equality, nothing changed.
- *Get the test.* The process is described visually in figure 2.11. If the methods are different or the query explicitly mentions that even not changed methods should be used, then for each of the methods the index is obtained. Armed with the indexes, the XML mapping file is queried for nodes that have the “MethodID” attribute between the indexes. Once the nodes are identified, the test methods corresponding to their nodes are obtained and the result is returned.

2.6 User interface

TestNForce is designed to be simple. The interface consists of three menu items and a tool window. The menu items, shown in figure 2.12, are described below:

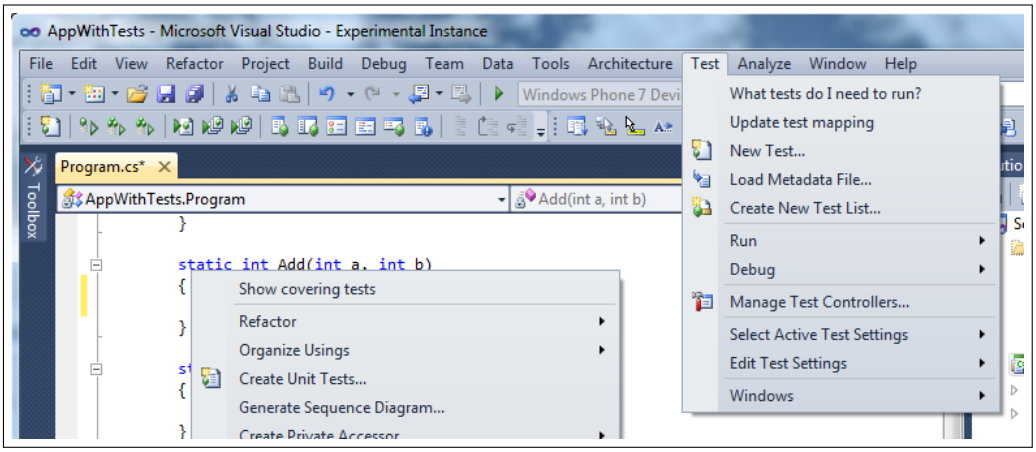


Figure 2.12: The three menus of TestNForce

- *Update test mapping*. This menu item, is placed in the `Test` menu of Visual Studio. Its purpose, as the name suggests, is to update the test mapping index with the latest changes in the code. When invoked, it will trigger the update process in TestNForce and will display the status and output in an output window, just like the compilation process does.

- *What tests do I need to run?*. Placed next to `Update index` in the `Test` menu, this menu item displays the tests that need to be run based on the changed code. All the files in the solution are analyzed so the reason of placing this menu item in a toolbar menu is that its action is not context sensitive.

- *Show covering tests*. The only context sensitive menu of TestNForce, it is enabled only when right clicking text in C# files in the code editor. It will display the lists of tests that cover the line of code on which the right click was performed. Results will be displayed even if no changes were made to the code. Because of the reasons presented in section 3.1, when this menu item is invoked on a line of code inside a method, the option will be invoked for the whole method.

The tool window that presents the covering tests has just a listbox(figure 2.13). It shows the full name of the each test that was obtained using one of the menu items previously mentioned.

It might sound that the interface of TestNForce is too simple. However, TestNForce just provides the list of test cases that need to be executed and that is all what is needed for this.

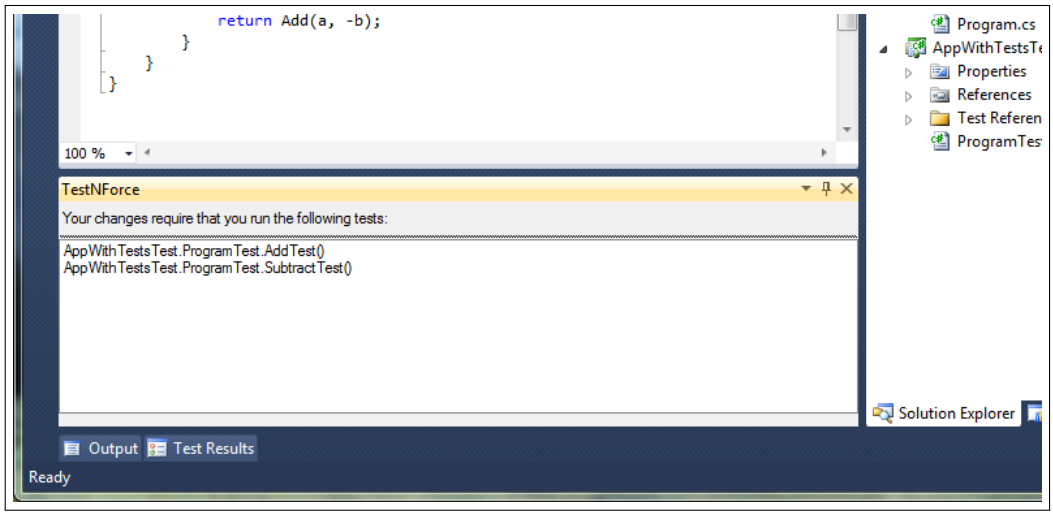


Figure 2.13: Test results as shown in Visual Studio

As mentioned in section 3.1, the interface could be simplified even more if a background analyzer would look for changes all the time.

2.7 Team Foundation Server checkin policy

TestNForce offers a checkin policy[4] for Team Foundation Server. The policy will prevent checkins if the test mapping index is not up to date.

The policy integrates with the standard checkin window (figure 2.14) and for those unfamiliar with Team Foundation Server, it not might be obvious which part of the window is the actual addin. We strongly believe that the previously mentioned fact is a plus for TestNForce because if keeps a unified experience across the development process.

Two types of warnings are displayed:

1. *The test mapping index is not included in the checkin*. The index must be included with the checked files because other developers should be able to get the updated version. This policy is triggered only when C# files are included in the checkedin files list.

2. *The test mapping index is not up to date*. When C# files are included, the index file must be the latest updated file. Otherwise, the index is not up to date and the policy is triggered.

[4] As described on MSDN: "Check-in policies are a mechanism for enforcing development practices across your development team."

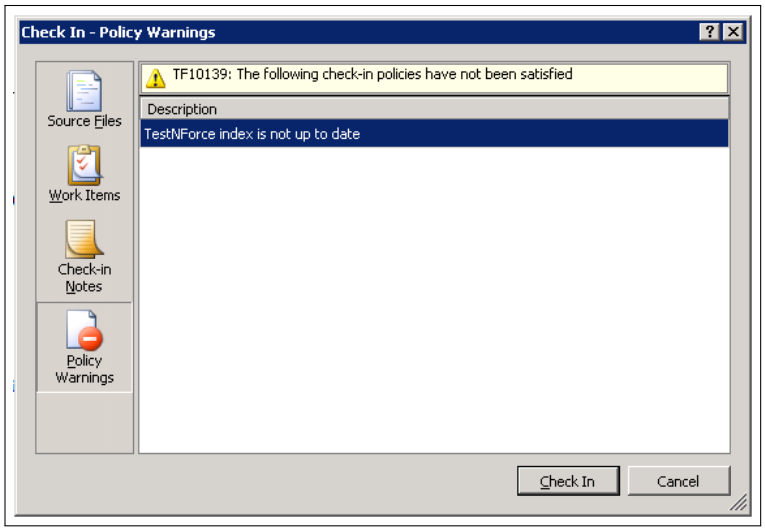


Figure 2.14: The checkin policy

2.8 Summary

In this chapter, the technical details of TestNForce were presented. The implementation is far more complex and only the most important aspects were detailed.

In the next chapter, we'll see if the current implementation of TestNForce can help developers and testers.

Chapter 3

User study and survey

During the development of TestNForce, a survey and an user study were conducted. The first one was informal and it had the purpose of improving the usability of TestNForce while, the latter one, took place in a controlled environment and had the purpose of evaluating the final version of TestNForce. This chapter is about them.

Each of the two is presented in two parts. The first part always gives details about the setup and execution while the second part carefully analyzes the outcome.

3.1 The usability survey

Just after the start of the Visual Studio integration implementation phase, a usability problem arose. The question without answer was "What should be displayed when invoking the "tests that cover this code" option from the context menu in editor?". This question was and still is open ended. More specifically, a number of subquestions need to be answered: What is displayed when the previously mentioned option is invoked for ...

1. ... a few lines in a method are selected?
2. ... the whole method is selected?
3. ... one or more methods are fully selected and one or more other are partially selected?
4. ... a namespace is selected?
5. ... an empty line in a method/class is selected?
6. ... the "using" section is selected?

For example, the first subquestion, "What happens when a few lines in a method are selected?" can have multiple answers. The user can get only the tests that cover those specific lines; in this case, if we have an `if` block and only the `else` branch is selected, any test that covers the `if` branch will not be displayed. Another solution, which is as valid as the previous one, could give the user all the tests covering the method in which the lines are;

in this case we would have a *method level granularity*[1]. A third alternative, but definitely not the last one possible, is to display either the tests covering each line in the selection or just an aggregated result when branches exist.

We had multiple answers to these questions and could not decide what the end users will find more useful. In order to answer the question and to make the TestNForce experience very pleasant, an informal survey was conducted. A number of six subjects were selected. They all work at Microsoft and have extensive experience with Visual Studio, which is the primary IDE used in the company. There was no reason to collect background information about participants because they all have a similar profile and no major differences could be observed.

They were all gathered in one room where they filled the questionnaire while the author was showing, on a projector screen, different methods of selection and menu invocations.

The questionnaire had 4 questions. The first one covered the subquestions previously mentioned. The second question was asking about the expectation when just the method signature/class name/namespace name is selected and the option is invoked; is it expected to have results for each line of code or just one aggregated result? The third question was asking about expectations for methods/classes/namespaces that split over multiple files (C# supports partial methods since version 3.0[2]). The final question was asking the participants to write their ideas and to give details if they see a better approach for the selection method.

The questionnaire can be seen on page 62.

3.1.1 Results

The answers to the first question were aligned to our initial belief. If the code file is seen as a tree with lines of code as leafs, having the method as parent, having the class as parent, having as namespace as root then, selecting a node will invoke the action on all the leafs. For empty lines, the action will be invoked on the node that contains the empty line.

For the second question, only one participant thought that the aggregated results are enough. All the rest thought that the aggregated result is good but there should be an option to expand the result for details.

An unanimous answer was provided for the third question: when selecting a namespace that splits over multiple files, all the files should be analyzed. For any other code block (class, method), only the local context should be analyzed.

After a brainstorming session for the fourth question, the following ideas were produced:

1. "A popup dialog that provides different options can be shown prior to results"

2. "Automatic discovery while typing": the index will be queried while changing the code and results will be updated in real time. While this solution is very interesting and would simplify the interface (there is no need for menus anymore), it might be

[1] This means that when details about the content of a method are asked, only details about the method itself can be obtained

[2] `http://msdn.microsoft.com/en-us/library/wa80x488\%28v=VS.100\%29.aspx`

impossible to implement it for large projects because of the time required to query the index.

3. "Code overlay, like code coverage does": in Visual Studio, the code coverage result is displayed over the code, by coloring lines with different colors. Initially, it was not clear how such a solution can be implement. The answer was later clarified, verbally, by the participant: display, for each line of code, on its left (where the breakpoints are displayed) a rectangle that shows a popup window with tests that cover that method, when clicked.

Only part of the ideas extracted from the survey's results were used in the final design. This is mostly because the initial design had a line of code granularity for the tests that cover a specific block of code. The final design, because of some limitation of the Visual Studio API, allows only method level granularity (the finest granularity that can be obtained is at method level; when an option is invoked for elements inside a method, the result is equivalent with invoking the option at method level).

3.2 User study

A tool is useful if it solves a real problem and it is not creating new ones. TestNForce might change the way software development is done and, in some cases, the change might be radical. It is dangerous to say that TestNForce is good just because it tries to solve a real problem. A proof was needed.

A historical analysis of tools similar to TestNForce, in order to decide what is good/bad by looking at the past was not a feasible option. To the best of our knowledge, only one tool, the Test Impact Analyzer from Visual Studio 2010[3], is available and it does not have enough history behind. Therefore, TestNForce was evaluated through the meanings of a user study (organized as an experiment).

The first part of the section describes the experiment and the way it was prepared. The section ends with the outcome of the experiment. Also, conclusions about the usefulness and usability of TestNForce are drawn at the end.

3.2.1 Experimental design

Great attention must be given to the experiment design because only an experiment that is well designed can give high quality results. The efficiency of the analysis of the results depends on the design of the experiment that was conducted to extract the results[6]. Therefore, the experiment was carefully designed taking into account, from the beginning, the questions that must be answered.

A one-shot case study[4] would be of little use in the case of TestNForce because the group is only observed once while "securing scientific evidence involves making at least one comparison[4]". A static-group comparison design would give very good data because two groups, one using TestNForce and one not using it could be observed and their results

[3] http://msdn.microsoft.com/en-us/library/dd264992.aspx

compared. If designed and planned correctly, the only difference between results would be caused by TestNForce. However, as will be mentioned in section 3.3.2, the experiment involved a number of participants insufficient for making more than one group.

A *one-group pretest-posttest design[4]* was selected for evaluating TestNForce. Even though such a design is categorized as a pre-experimental design, for the rest of the paper we will to it as "experimental design" and "experiment", respectively. The one-group pretest-posttest design involves a group that is observed before and after some (controlled) variables are changed. In the case of TestNForce, the only change between the two observations is the usage of the tool. A number of subjects were asked to use TestNForce and express their opinion before and after using it.

In order to reduce the effects of history[4], meaning that the results could be altered by time if the time difference between the pretest and the posttest spans across a long time interval, it was decided the run the whole experiment without breaks. The experiment was preceded by a pilot run which highlighted some problems with the tool and with the experiment environment.

Before designing the experiment material, a number of variables, that will be observed, need to be defined. The variables chose for analysis are:

1. *Usefulness*.[4] The degree to which TestNForce can solve a real problem and is helpful.
2. *Usability*.[5] The degree to which TestNForce is easy and pleasant to use.
3. *Completeness*.[6] The degree to which TestNForce solves the problem or how much of what is necessary was left uncovered.

Having the values for the above mentioned variables, we can answer a part of the research questions mentioned in 1. The first and the third variable provide the answers to research question 3 while the second and, again, the third variable give the answer to the forth question. The third variable has a double role because an incomplete product reduces its usefulness and frustrates the users. To summarize, the two research question that will be answered by the experiment are:

1. Is TestNForce useful?
2. Is TestNForce providing a good user experience?

3.2.2 Pretest and posttest

The pretest

The purpose of the pretest was to gather some background information about the subjects and to get their expectations from a tool like TestNForce. Four categories of questions were

[4]Usefulness = (1) Having a beneficial use; serviceable (2) Being of practical use
[5]Usability = (1) That can be used (2) Fit for use; convenient to use
[6]Completeness = Having all necessary or normal parts, components, or steps; entire

asked. The questions in the first three categories collect relevant information about the participant in order to create its profile. The questions in the last category collect information about their expectations from a tool like TestNForce.

1. *Participant's background (questions 1a-1d).* Questions about age, profession and education were asked

2. *Development experience (questions 2a-2i).* Subjects were asked to rate their development skills in general, development skills and experience with C# and Visual Studio and experience with large projects. The information collected can be used to determine if the answers of the participants differ with the experience level.

3. *Testing experience (questions 3a-3h).* Because TestNForce involves testing and tests, information about the testing skills was collected using the questions in this category. Participants had to rate their testing skills, their expectations from tests (expected code coverage, expected tests for small projects, time spent on testing) and also had to share what kind of test, if any, they did in the past.

4. *Expectations from TestNForce (questions 4a-4f).* The participants read an abstract description of TestNForce (printed below) and expressed their expectations with regard to usability and usefulness of such a tool. Also, they had the opportunity to write if the ever heard and/or used a tool that matches the description.

   > "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."

The full pretest can be seen on page 63.

The posttest

After the assignment was done or the time allocated for it elapsed, the participants had to fill a second questionnaire. This time, the questions were directly targeting TestNForce and the experiment. Seven categories of questions were created. Following them, an additional open ended question was asked and the participants had the chance to express any thoughts that were not covered by previous questions.

The seven categories are as follows:

1. *Simple questions about TestNForce (questions 1a-1c).* The subjects were asked if they liked TestNForce and if they found it easy to use.

2. *TestNForce in relation to the assignment (questions 2a-2c).* The questions in this category were asked in order to figure out if the tool helped the participants solve assignment.

3. *TestNForce and Team Foundation Server (questions 3a-3d).* The assignment covered the integration between the Team Foundation Server and TestNForce. The participants had to express their opinion about the integration. The questions asked were about the quality of the integration, about the usefulness of the integration and about usability.

4. *Usability (questions 4a-4f).* A tool must not be difficult to use. Questions in this category are meant to provide a clear answer about TestNForce: is it easy to use it in general?

5. *What is missing (questions 5a-5f).* We are aware of some features that were planned initially as "nice to have" but, the lack of time removed it from the backlog. These features are listed in this category and participants had to tell, for each of them, if they would like or not, to have them. The features includes support for other programming languages, static code analysis, incremental index update, etc.

6. *Assignment (questions 6a-6e).* Even though the evaluation of tool was the primary goal of the experiment, the assignment had to be evaluated. Important conclusions can be drawn from these questions. Having the answers to questions like "Was the time allocated for the experiment enough to complete the assignment?" or "Was there enough guidance?" can tell if the focus of the subjects was on the tool and on the assignment or they just tried to overcome different blocking issues, not related to the experiment.

7. *Experiment (questions 7a-7f).* Just like the questions in the previous category, the experiment questions are meant to decide if the tool was evaluated well enough or the experiment actually focused on irrelevant aspects. The subjects were asked if the experiment was well organized, if they think that Jurassic was appropriate and they were asked to rate their overall impression of the experiment.

The full posttest can be seen on page 65.

Some of the questions in the posttest are the same as the ones in the last category of the pretest. This allows the comparison of expectations before using TestNForce and the actual impression after.

All the questions, except the last one, can be answered with one of the following 5 answers, each answer having a score value associated with it: (1) Totally disagree, (2) Somehow disagree, (3) Neutral, (4) Somehow agree, (5) Totally agree.

3.2.3 The assignment

Even though it is presented after the posttest, the completion of the assignment took place, chronologically, just after the pretest and before the posttest. The experiment took place in Delft, the Netherlands while the author of this thesis lives in Copenhagen, Denmark. Because of this and, because each participant is volunteer and does not have too much time for the experiment, it was decided to run the experiment over 2 days. Based on the number

of participants, the time allocated for the assignment was 1 hour, with a maximum extension of 5 minutes in special situations.

The assignment was designed in such a way that it will require the participants to use TestNForce but, will still be a realistic scenario. There was a total of nine participants that completed the assignment (one in the pilot and eight in the real experiment).

The content

The assignment is constructed around a semi-realistic scenario. The subject is a newly hired developer at Monroe Corporation. He is arriving at work on his first day and everyone is away. He finds only a note (the assignment) that mentions that everyone else is in Bahamas for a team building event and he is told that this should not stop him from starting. The note gives him a brief introduction of Jurassic and TestNForce. The final paragraph asks the "employee" to not spend too much time at work in the first day and mentions that 60 minutes are enough. This last paragraph actually sets the time limit for the experiment.

The first assignment is asking the subject to get familiar with the (test) code. It is important for a developer/tester to know which tests are covering a specific method. First, the participant has to spend a few minutes trying to identify the tests manually and then he is asked to use TestNForce to achieve the same result.

The second assignment is about change risk. The assignment states that the risk of a change increases with the number of affected tests. This holds as long as there is enough coverage and, luckily, this is valid for Jurassic. The participant has to decide if adding a new base type to the compiler core can break many 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".

The third question was the first and only, programming assignment. The request 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 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.

The forth assignment and the fifth could be done together. Actually, the fifth is a consequence of the forth and can be involuntarily completed while doing the former. The forth assignment asks to checkin the changes that the subject did. This is not possible until the index is updated and added to the project so the fifth assignment can be completed here. However, updating the index takes 28 minutes so, participants were asked to start the process, but not wait until is finished.

Jurassic

The assignment had to be created around a project. A number of criteria had to be satisfied in order for a project to be a good candidate (in random order):

- The project has to be free or its license should allow the use in academic environments

- The full source code must be available
- It should not be graphically intensive because it will run on virtual machine, where 3D acceleration is not available
- It should have a considerable number of test cases that pass and a good coverage
- It should not be a trivial application
- The tests cases must take more than a few minutes to run
- The code used should be C# only
- MSTest should be the test platform
- It should be either known by everyone or by no one in order to have a fair experiment

The selection process was extremely difficult. Most projects found satisfy just partially the conditions stated above. The biggest community of .NET open source projects is CodePlex[7] and that is, to the best of our knowledge, the only place where such a project can be found.

Between the tools that satisfy most of the conditions we find tools like Ajax Control Toolkit[8] - "a rich set of controls that you can use to build highly responsive and interactive Ajax-enabled Web applications" - Microsoft All in One Code Framework[9] - "The Microsoft All-In-One Code Framework is a free, centralized code sample library driven by developers' needs" - or StyleCop[10] - "StyleCop analyzes C# source code to enforce a set of style and consistency rules".

The previously mentioned projects have the disadvantage of being well known between .NET programmers but, probably, unknown to others. It would have been possible to have some experienced .NET developer taking part in the experiment and his results would, mostly sure, not be comparable with other's who do not know the code base.

During the "Most popular projects" list interrogation, Jurassic showed up. Jurassic is "an implementation of the ECMAScript language and runtime. It aims to provide the best performing and most standards-compliant implementation of JavaScript for .NET. Jurassic is not intended for end-users; instead it is intended to be integrated into .NET programs. If you are the author of a .NET program, you can use Jurassic to compile and execute JavaScript code.[11]".

Jurassic satisfies all the prerequisites for the project and is worth mentioning that a number of 344 test cases are available, out of which 328 pass. The project is extremely complex since it translates JavaScript code to Microsoft Intermediate Language but, still is very simple to learn due to a good architecture. This makes it the perfect candidate.

[7] http://codeplex.com
[8] http://ajaxcontroltoolkit.codeplex.com/
[9] http://1code.codeplex.com/
[10] http://stylecop.codeplex.com/
[11] http://jurassic.codeplex.com/

The only drawback: the Jurassic solution has a Silverlight project. TestNForce does not support XAML code. For this reason, the project was removed from solution but, the functionality of Jurassic was not impacted at all since the Silverlight project was just an user interface for demo purposes.

3.2.4 The pilot run

After the design of the assignment was completed, but before the actual experiment run, a pilot run was set up. Such a run was scheduled in order to be proactive and catch any unexpected issue. The run was schedule approximately 2 weeks before the final experiment date. This time interval was considered sufficient to fix any issue that might surface during the pilot.

Two major issues were discovered and fixed subsequently. The first bug caused the index not to be updated correctly because the virtual machine had an outdated version of TestNForce. This issue was mitigated by copying the latest version of the binaries and updating the index.

The second issue was caused by the fact that Visual Studio is not caching the credentials for the Team Foundation Server in the experimental instance if they were typed in the non-experimental instance. This leads to the impossibility of connecting to TFS without assistance - the participants could not know the URL of the server. This problem cannot be fixed and it was decided that each participant to ask for the credentials and URL before checkin.

Other minor issues fixed after the pilot experiment:

- Added more hints for the first assignment, especially regarding the Find References features of Visual Studio
- Concluded that a crash course on Visual Studio is needed, if the participant has no prior experience because, the pilot participant had some troubles finding its way through the IDE
- Fixed a typo in one of the menus (“index” was written as “idnex”)
- Fixed some grammar mistakes in the assignment’s text

3.3 The experiment run

In the graphical representation of the results, the horizontal axis which contains numbers from 1 to 8 always represents the subjects’ answers, each bar corresponding to the candidate with that index. The vertical bar with values from 1 to 5 is always the value of the response (1 = Totally disagree; 2 = Somehow disagree; 3 = Neutral; 4 = Somehow agree; 5 = Totally agree). Questions starting with “pre” are part of the pretest while those starting with “post” are part of the posttest.

3.3.1 Experiment environment

In an attempt to mitigate possible failures caused by the software policies in the TUDelft network, a virtual machine was set up. Since all the technologies used during the development of TestNForce were signed by Microsoft, the machine emulation software could not be signed by someone else.

The operating system of the virtual machine was Windows 7 Ultimate. All the unnecessary components (Media Player, themes, Media Center, games, etc.) were removed and only Visual Studio Ultimate with Team Explorer was installed. There was no restriction on what applications can be used but, as expected, there was no need for any other software, not even for the Internet browser. The Team Foundation Server was running on another Windows 2008 machine. Participants did not had direct access to this machine and their only interaction with the source control server was though Visual Studio.

The configuration for which TestNForce was created and on which it was tested has the characteristics of a netbook. Such a configuration includes a slow processor (usually single core with a maximum frequency of 1.6GHz and 1-2GB of RAM memory). All the computers used in the experiment have a better configuration that the recommended one, so there is no doubt that the hardware could not have impacted, in a negative way, the results.

Upon start, participants were asked to run a small script that will clean the machine (if there were any leftovers from the previous experiment run) and setup the environment for them. Each participant had a fresh, untouched copy of TestNForce, Jurassic and experiment documents.

3.3.2 Participants selection

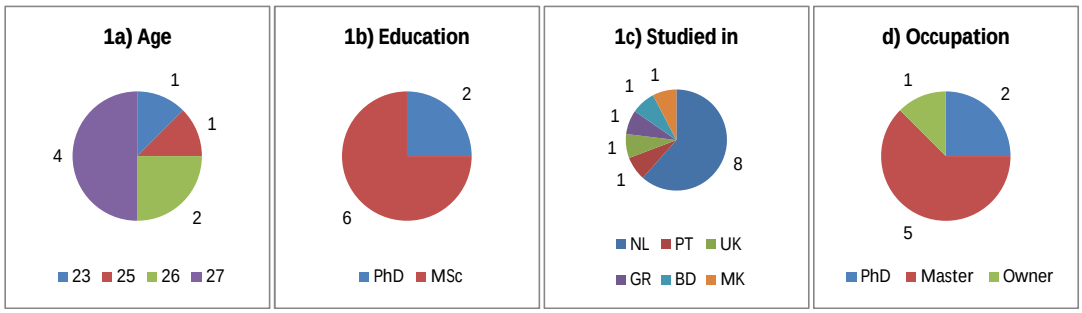
In order to evaluate TestNForce, the ideal subject would have some background with .NET technologies and would have worked on projects that have many tests and require a long time to run. Such a profile is required in order to be able to evaluate if TestNForce improves the development process - without prior knowledge it is hard to say if things can be different without the tool.

Even though Java is the primary language used in the University of Delft, there is still a good number of students and staff members that have .NET knowledge. The experiment invitation was sent without taking into account the subject's profile. Later, it was realized that not including a prerequisite was a enormous risk because the final subjects could be completely unfamiliar with .NET. However, it turned out that most of the participants had at least basic knowledge of Visual Studio and .NET and they worked with tests before.

The invitation was sent to approximately 20 individuals out of whom eight confirmed their availability in the proposed time intervals. All who responded were selected for participation.

3.3.3 Participants' profile

As mentioned before, a number of 8 participants took part in the experiment. This section gives an overview of their profile based on their answers from the pretest.



(a) Answers to question *pre1a* (b) Answers to question *pre1b* (c) Answers to question *pre1c* (d) Answers to question *pre1d*

Figure 3.1: Background information about subjects

Participant's background

The participants have the age between 23 and 27 years, most of them being situated in the upper part of the interval as can be seen in figure 3.1a. Two of them are PhD students (phd), five of them are master students (msc) and one of them is owner of an IT consultancy company (own) (figure 3.1d). Figure 3.1b shows that their educational background is similar, all having or being in the process of getting academic degrees. All the subjects studied in Netherlands (NL). Moreover, just three of them studied in one country only, the rest being equally distributed between: Portugal (PT), Greece (GR), Macedonia (MK) and Bangladesh (BD). A graphical representation of this data can be seen in figure 3.1c.

Development experience

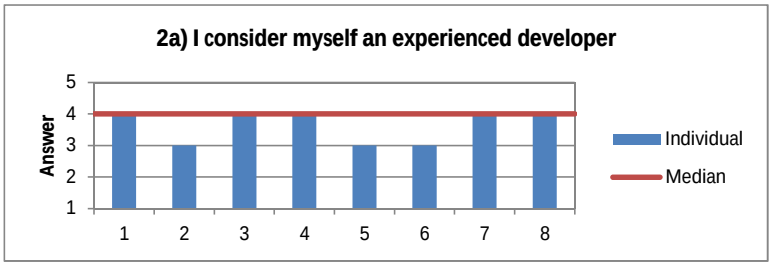
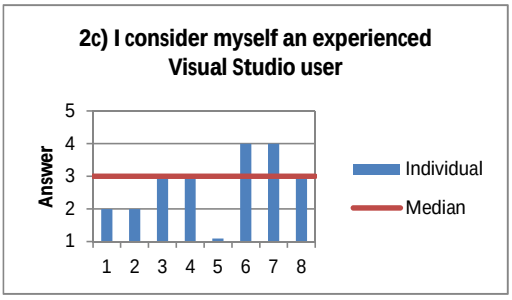


Figure 3.2: Answers to question *pre2a*

All participants consider themselves at least average experienced developers. Figure 3.2 shows the previously mentioned observation and also shows that none of the participants is a guru. For evaluating TestNForce such a skill level is almost ideal because the majority of the developers in the field have similar skills and extremes are rare.

2b) What is (are) the development environment(s) that you are experienced with?									
IDE	#	1	2	3	4	5	6	7	8
Borland	1	✓	✗	✗	✗	✗	✗	✗	✗
Eclipse	8	✓	✓	✓	✓	✓	✓	✓	✓
Netbeans	1	✗	✗	✗	✓	✗	✗	✗	✗
Vim	2	✗	✓	✗	✗	✓	✗	✗	✗
VS	5	✗	✗	✓	✓	✗	✓	✓	✓

(a) Answers to question *pre2b*



(b) Answers to question *pre2c*

Figure 3.3: Experience with IDEs

Figure 3.3a shows the Integrated Development Environments with which each participant is familiar. It can be seen that all subjects have knowledge of Eclipse and more than half of the participants have worked with Visual Studio (VS). Having an Eclipse background makes the transition to VS simple because the two IDE have many similar features. None of the participants expressed difficulties in using VS and there is no reason to believe that the VS knowledge affected the experiment results. On the official website of VIM is it stated that "[Vim] is often called a programmer's editor, and so useful for programming that many consider it an entire IDE[12]". The inclusion of VIM in the list might be arguable by some. No matter what the correct categorization is, the answer will stay in the list in order to not alter the subject's profile.

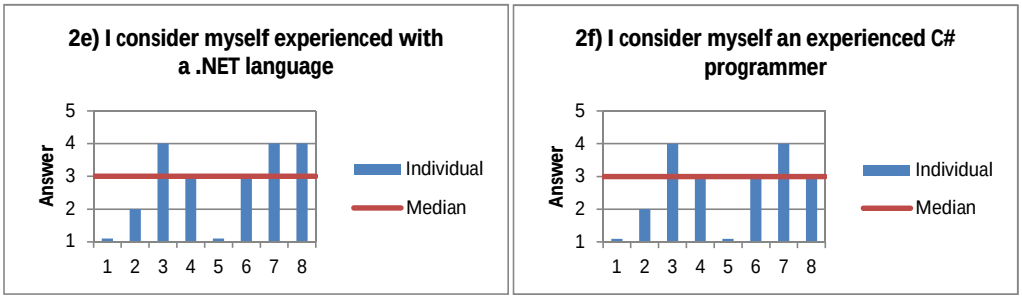
In figure 3.3b, the experience of each participant with VS can be seen. These answers are strongly connected to those of question *Pre2b*. The participants that did not include VS as an answer in *Pre2b*, answered with "Strongly disagree" or "Somehow disagree" in question *Pre2c*. An important observation is that only one participant (number 5) has no experience with VS; all the others have at least some.

2d) What is (are) the programming language(s) that you are experienced with?									
Language	#	1	2	3	4	5	6	7	8
C	5	✓	✓	✗	✓	✓	✗	✗	✓
C++	3	✓	✗	✗	✗	✗	✓	✗	✓
C#	4	✗	✗	✓	✗	✗	✓	✓	✓
Haskell	1	✓	✗	✗	✗	✗	✗	✗	✗
Java	8	✓	✓	✓	✓	✓	✓	✓	✓
R	1	✗	✗	✗	✗	✓	✗	✗	✗
Python	3	✗	✓	✗	✓	✓	✗	✗	✗
Stratego/XT	1	✗	✗	✗	✗	✓	✗	✗	✗

Figure 3.4: Answers to question *pre2d*

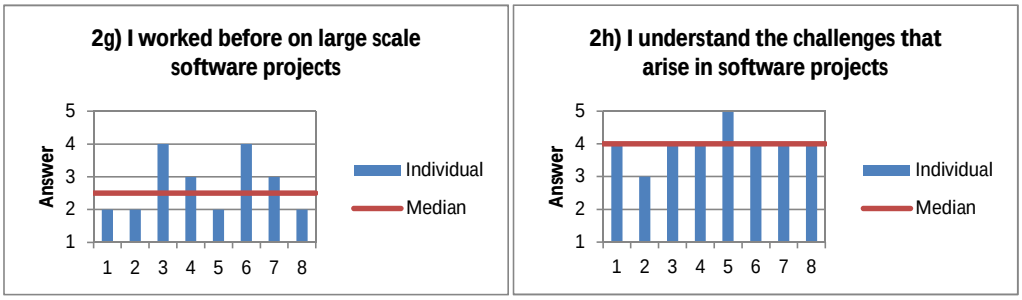
The information collected about subjects' background was their experience with differ-

[12] http://www.vim.org/about.php



(a) Answers to question *pre2e* (b) Answers to question *pre2f*

Figure 3.5: Experience with managed languages



(a) Answers to question *pre2g* (b) Answers to question *pre2h*

Figure 3.6: Experience software projects

ent programming languages. The table in figure 3.4 shows the answers. Since the participants are all students at the Technical University of Delft (see answers from question *pre1c*) where Java is one of the most common programming languages, the answers reflect this aspect. All the participants have experience with Java and half of them know C#. Moreover, some participants have C++ knowledge. Having such a good background with OOP languages, makes even the subjects with no C# experience able to program in .NET.

Figures 3.5a and 3.5b show the experience of the participants with a .NET language (which could be C#, VB.NET, F#, C++/CLI, IronPython, etc.) and the experience with C#, respectively. The results of these two questions can give clues about the problem, if the subject has difficulties in completing the programming assignment. It can be observed that two participants (1 and 5) have no prior experience with managed languages. Subject 8 mostly sure knows another .NET language better than C#. The rest of the participants have the same answers in both questions which lead to the conclusion that the language experience expressed in question *pre2e* is actually C#.

Based on the results in figure 3.6b it can be concluded that participants know what are the pain points of software projects and they will be able to evaluate TestNForce correctly and decide if it helps them in delivering faster/better products. However, the answers in

figure 3.6a shows that while the participants understand the challenges, most of them did not work in large projects. This might have been a threat to the validity of the experiment because the subjects could have only theoretical knowledge about the projects that TestNForce targets. However, the answers in the post test show that participants actually identified correctly the benefits of TestNForce.

None of the participants heard of a tool similar to TestNForce (question *pre2i*), therefore there is no reason to discuss this answer. The only observation is that the subjects are equal from this perspective and their answer will not be affected by the previous experience.

Testing experience

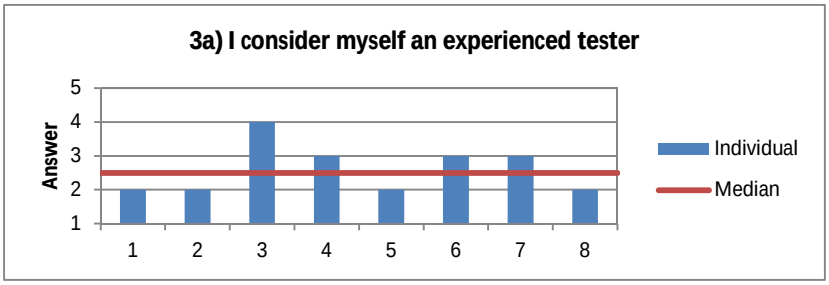


Figure 3.7: Answers to question *pre3a*

Figure 3.7 shows how each participant evaluated his testing skills. If the results are compared with those from question *pre2a*, an interesting pattern will be observed: all participants have a lower (or equal) testing skill.

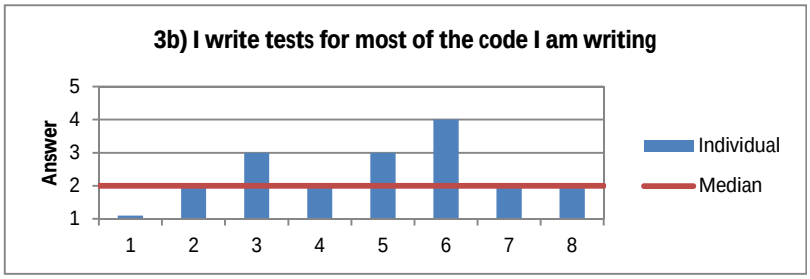


Figure 3.8: Answers to question *pre3b*

The responses to question *pre3a*, most probably, are influenced by the responses to question *pre3b*. In figure 3.8 it can be noticed that participants do not really write tests for the code they write, with one exception. An interesting response was provided by subject 8 who mentions that he is writing some tests but, as will be seen in the responses of question *pre3c*, no test type was mentioned. Even though the participants understood correctly the

purpose of TestNForce and identified the problems it is solving, it might be argued that the lack of test experience might have impacted the test results.

3c) What kind of tests you did in the past?									
Test type	**#**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Acceptance	1	✗	✗	✗	✗	✗	✓	✗	✗
Integration	2	✗	✓	✗	✗	✗	✓	✗	✗
Manual	1	✓	✗	✗	✗	✗	✗	✗	✗
Unit	6	✗	✓	✓	✓	✓	✓	✓	✗

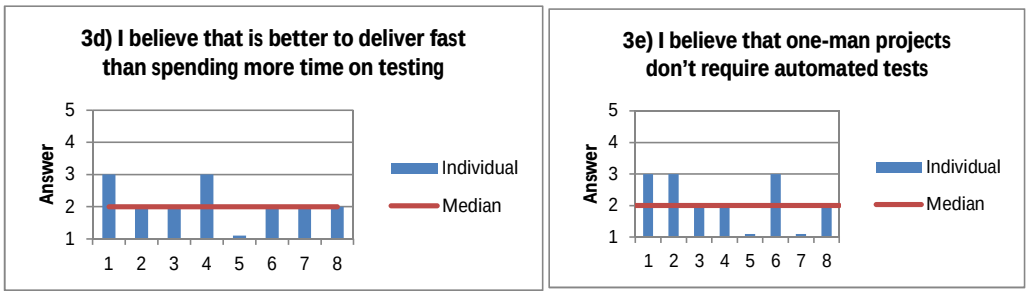
Figure 3.9: Answers to question *pre3c*

Question *pre3c*, which has the results presented in figure 3.9, asks the participants, indirectly, to give more details about the response from the previous question. The participants wrote what kind of tests they did in the past. It can be noticed that most participants wrote unit tests which are the tests that TestNForce targets.

The questions *pre3d* to *pre3h* are used just to get an idea about the level of expertise of the participants and their work style. From figure 3.10a it can be deduced that the participants do not want to deliver low quality code but, unfortunately, it cannot be deduced if they would have the same opinion under pressure (time or budget pressure). This question is valuable for understanding the profile but its design is incomplete.

On the other hand, the results in figure 3.10b, reflect the style of the participants when working alone. We believe that the responses reflect the approach that the participants would take even when working on a multi-man project, where there are no test criteria. It can be seen that three of the participants said that there is a 50:50 chance for them writing tests. However, in the post experiment discussions, many participants admitted that, unless told to, they do not write tests.

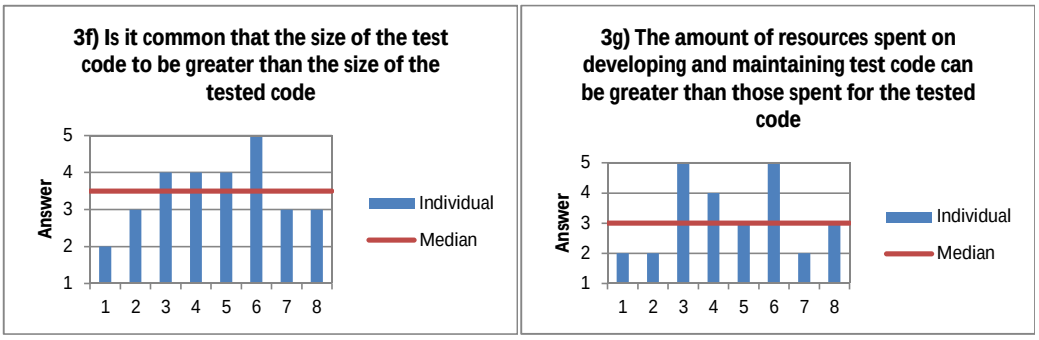
The subjects had to estimate or tell from experience their expectations for the resources involved in testing. The responses in figure 3.11a and 3.11b shows that, on average, the participants are aware about the fact that the size of the test code can be greater that that



(a) Answers to question *pre3d*

(b) Answers to question *pre3e*

Figure 3.10: Experience with testing



(a) Answers to question *pre3f* (b) Answers to question *pre3g*

Figure 3.11: Estimation of test resources

of the code under test and that it can also be (the test code) more complex. Also, the participants are aware that the time spent on testing can be greater than the time spent on developing the code under test. The advantages and disadvantages of spending extra time are also presented by George in [12] where he conducted an experiment and showed that more resources are needed when testing is involved.

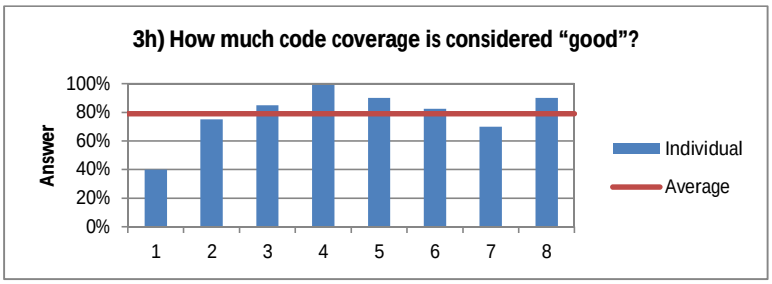


Figure 3.12: Answers to question *pre3h*

The last question in the testing background category was about code coverage. The participants provided the answers shown in figure 3.11b when asked how much code coverage is considered “good”. This question has the purpose of highlighting what the subject understands as code coverage benefits. A response of 100% means that the participant was not fully realistic because the return of investment of such a coverage is not justified. However, a too low value would mean that either the project is too small and too much time would be spent on developing tests or there are not enough tests. We believe that an 80% coverage is a realistic target, a target expressed by the average of the responses.

The profile of the participants makes them fit in the ideal subject profile. Just two of the participants indicated no prior C# knowledge and only one of them indicated no prior Visual Studio experience. All consider themselves average or above average developers and understand the challenges that arise in software projects. The only requirement that is not

fully satisfied by the profiles is the experience with large projects. Most of the participants indicated very little experience.

3.4 Experiment results and analysis

The ultimate goal of the experiment was two answer two research questions:

1. Is TestNForce useful?
2. Is TestNForce providing a good usability experience?

3.4.1 Evaluation of TestNForce

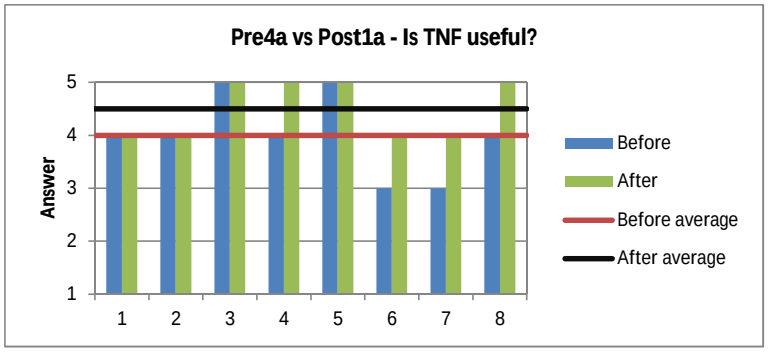


Figure 3.13: Before-after comparison of expectations about the usefulness of TestNForce

In the pretest, participants were asked to say if they would use a tool like TestNForce. In the posttest, they were asked if TestNForce was indeed useful. These two questions, even though phrased differently, try to evaluate if TestNForce is of any good. As can be noticed in figure 3.13, before having the chance to work with TestNForce, the subjects considered such a tool useful giving an average rating of 4 (of a scale from 1 to 5, 5 meaning "Totally agree"). The answers in the post test confirm the fact that TestNForce is useful. All the answers, after the assignment, remained either at the same level (which was already high) or improved. One important fact worth mentioning is that some subjects expected, during the pretest, that such a tool to be extremely useful (answering with the maximum 5) and it turned out that TestNForce has been at their expectation level.

In both the pretest and the posttest, participants were asked if the tool described and TestNForce, respectively, it is annoying. It is extremely important to have a tool that solves the problem and does not introduce new ones like difficulties in getting the answer. Most of the participants, as confirmed by their answers shown in figure 3.14, expected TestNForce to offer a good experience - the average rating for the annoyance level was 2.75, placing it in the lower part of the grading interval. Based on the answers in the pretest, it can be

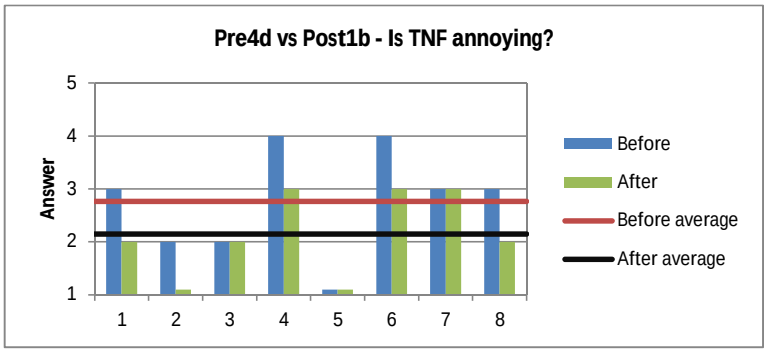
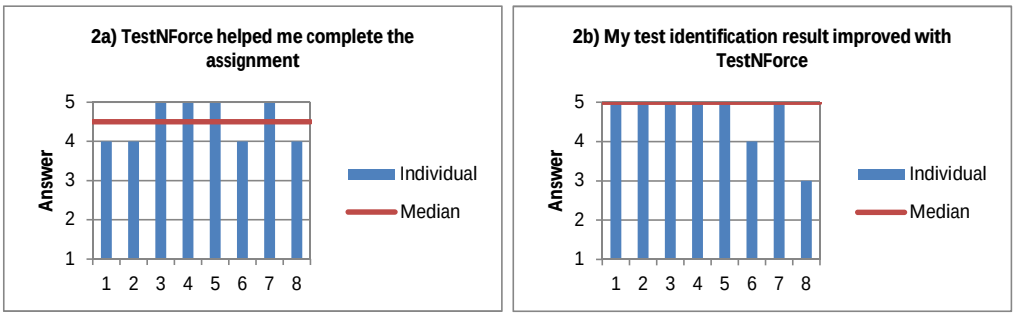


Figure 3.14: Before-after comparison of expectations regarding the fact that TestNForce might be/is annoying



(a) Answers to question *post2a*

(b) Answers to question *post2b*

Figure 3.15: Effects of using TestNForce

concluded that the expectations in terms of usability and annoyance were met. Even more, in 5 cases, the annoyance level was lower than expected.

Figure 3.15a, shows the fact that participants did used TestNForce to complete the assignment and the tool helped them. However, as one of the participants noticed, the assignment was designed in such a way that it will require the use of TestNForce. We agree with this statement but we do not believe that the assignment was structured to put TestNForce in a good light. The only reason why the assignment forced the participants to use the tool is that the only way of evaluating it is by using it. The answers to question *post2b*, shown in figure 3.15b are the reason of the answers to question *post2a*. The participants used TestNForce, their test identification skills improved (6 of the participants fully agreed with this statement), therefore the tool helped them complete the assignment.

None of the participants heard of Jurassic before the assignment (question *pre2i*). As can be seen in figure 3.16, TestNForce made them confident about changing unknown code which can lead to the conclusion that TestNForce does make the developer's job easier. The median rating of 4 strongly supports this. However, the participants might have been less

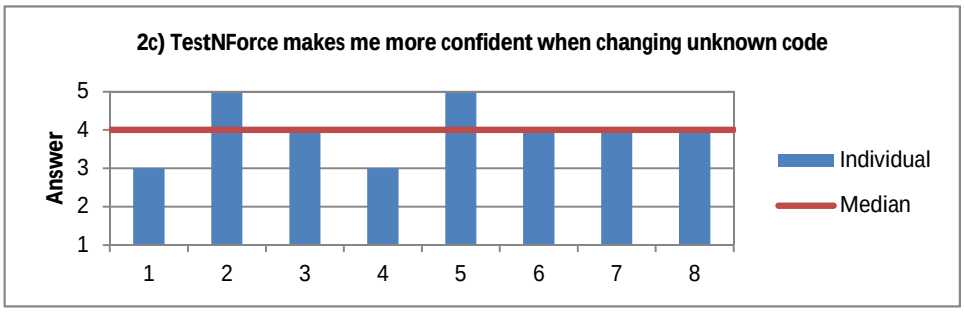
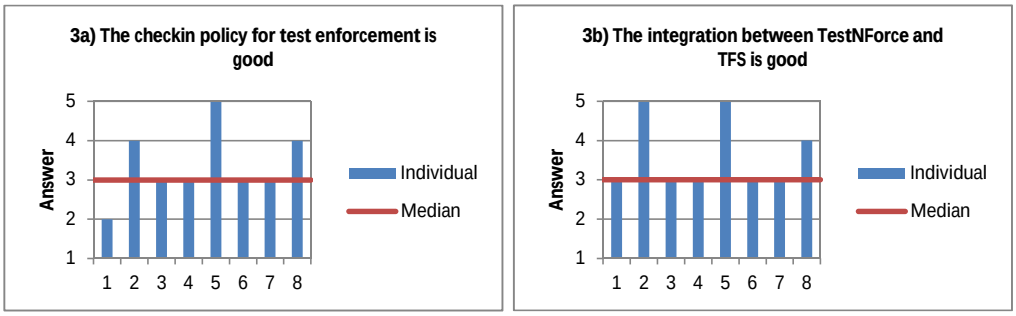


Figure 3.16: Answers to question *post2c*



(a) Answers to question *post3a*

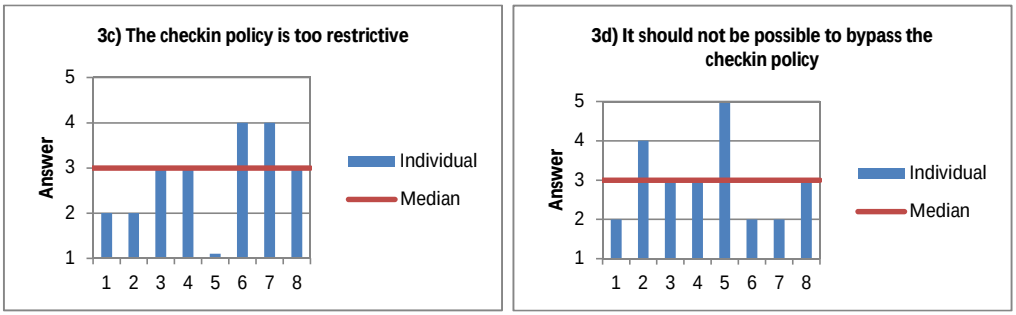
(b) Answers to question *post3b*

Figure 3.17: Evaluation of the checkin policy

worried about changing unknown code because they had explicit instructions and there were no consequences for mistakes.

From figure 3.17a it can be deduced that the checkin policy is mostly useful and most of the participants understood and appreciated its purpose. However, as can be seen from the results, there are not too many answers of "Totally agree" which means that no one considers the policy a critical component. The results presented in figure 3.17b represent the quality of the integration between TestNForce and Team Foundation Server (TFS). Because TestNForce integrates in the checkin window, there is no sign that could show that the policy is an external part of VS. This must be the reason why most of the participants considered the integration good. However, the integration is very light and for those who never seen the checkin window it might not show anything different; this might be the reason why some people gave a medium rating.

An interesting set of responses can be seen in figures 3.18a and 3.18b. Subjects 6 and 7, considered the policy quite restrictive (giving a rating of 4) but still they only "Somehow disagree" that the policy should have an option to bypass it (giving a rating of 2). On the other hand, subject 2 considers the policy very less restrictive but somehow agrees that it should be possible to bypass it. The rest of the participants think that the policy is mostly alright and it should not be possible to bypass it.



(a) Answers to question *post3c*

(b) Answers to question *post3d*

Figure 3.18: Restrictions of the checkin policy

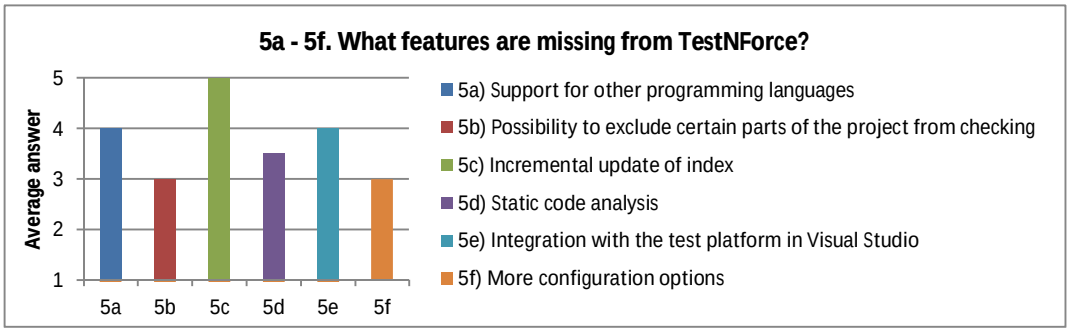
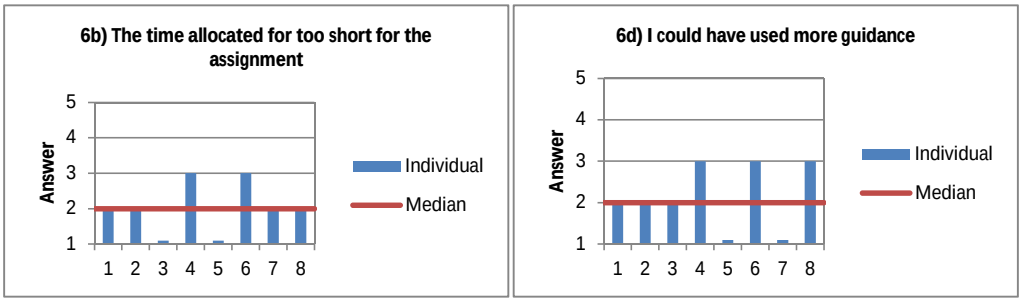


Figure 3.19: What should be added to TestNForce based on participant's opinion

All the questions in the fifth category of the post test ask about missing features. The results are presented in figure 3.19 as the average of the responses. Only the average is important here because new features are added based on what the majority of the users want, not on individual requests (individual responses can be seen on page 74). The most desirable feature is the incremental index update; this feature would allow TestNForce to update the index fast by replacing only the records that are affected by code/test changes. Following, the support for other programming languages and the integration with the test platform in Visual Studio are the requested features. The only missing feature, noticed by all the participants and not covered by the question, that must have impacted the usability, is the lack of navigation from the covering tests window to the actual test code. However, this was not a blocking issue because it could be worked around by searching the code and using the search results window for navigation. The backlog for future versions will prioritize the pain points identified by participants during the experiment. Another feature that was planned for the next releases and was also prototyped (but not used in the experiment) is a version of TestNForce that can work outside Visual Studio. The participants somehow disagree, on average, that there should be a standalone version (question *post4f*) therefore the priority was decreased, but this item is still on the backlog. We believe that a standalone



(a) Answers to question *post6b*

(b) Answers to question *post6d*

Figure 3.20: Assignment difficulty

version would allow the integration of TestNForce with different build systems.

3.4.2 Evaluation of the experiment

A number of 12 questions about the experiment were asked in the posttest.

Figures 3.20a and 3.20b give an overview of the difficulty of the assignment. The median value of 2 of the answers clearly shows that the assignment was not too difficult, but not trivial either which is exactly the point: the participants were able to complete the tasks, but they had to do a little effort for it.

One of the most interesting observations that can be derived from the information in the two previously mentioned charts is the fact that people with no Visual Studio experience found the assignment easier and required less guidance than those who mentioned no prior experience. The expectation would be that people with experience to find the assignment easier but as can be seen in figure 3.21, it was almost the opposite. There is no clear evidence why this happened.

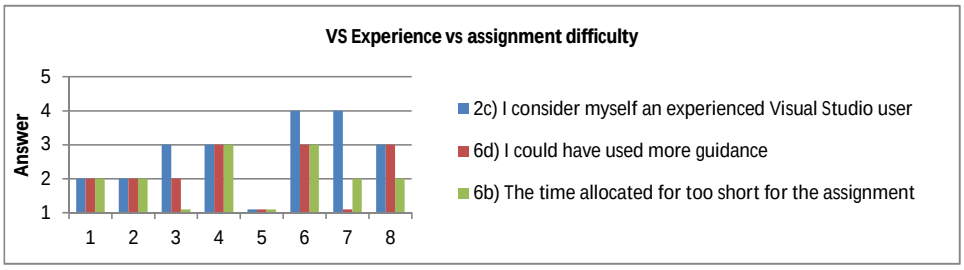
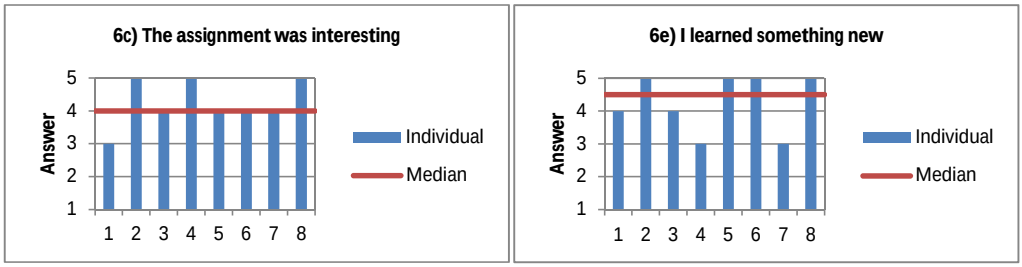


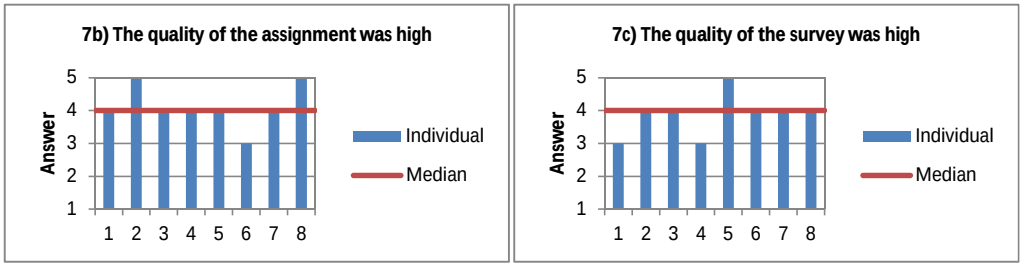
Figure 3.21: How difficult was the assignment compared to prior experience?

Figure 3.22a is a good indication that the assignment was interesting. Three of the participants rated the assignment with the maximum score, while the rest, except one, gave a rating of 4. The median rating of the assignment was 4. Also, four of the subjects strongly



(a) Answers to question *post6c* (b) Answers to question *post6e*

Figure 3.22: Benefits from the experiment



(a) Answers to question *post7b* (b) Answers to question *post7c*

Figure 3.23: Experiment impression

believe that they learned something new while just 2 of them were neutral when answered question *post6e*. The overall score for "I learned something new" was 4.5.

Both the assignment (figure 3.23a) and the survey (figure 3.23b) got a median rating of 4. This means that the quality was high but there was some place for improvement. The individual answers are all above the middle point (3).

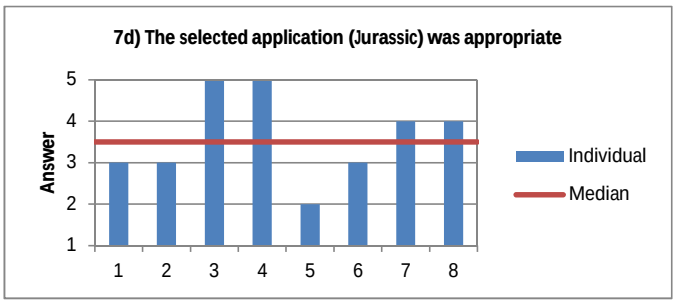


Figure 3.24: Answers to question *post7d*

From figure 3.24 we can conclude that Jurassic was not a bad choice but some were

not completely satisfied. One of the participants expressed his concern that Jurassic is a compiler and usually all the changes are high risk therefore, is not safe to assume, like the assignment did, that some changes might be low risk; a compiler either works or does not.

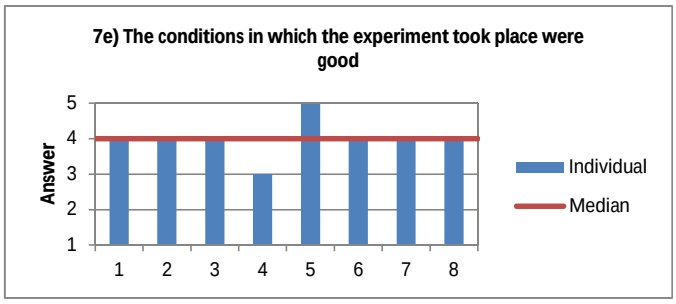


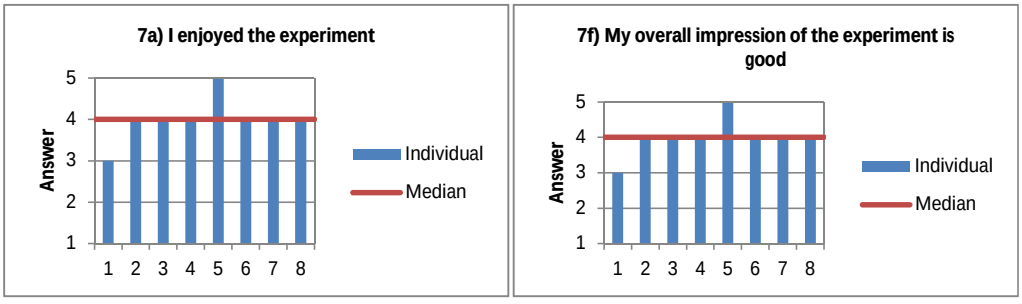
Figure 3.25: Answers to question *post7e*

We took great care in optimizing the conditions in which the experiment took place. The data in 3.25 confirms that the participants did not have any problems here. However, the conditions can be always improved. One of the factors that might have influenced the responses and made the participants not give the full rating is the fact that the experiment took place in a room where other students were working and there might have been distractions.

The median score of 4 in figure 3.26a shows that participants mostly enjoyed the experiment while a similar score for the overall experiment satisfaction shows, in figure 3.26b that the experiment was a success from the participant's point of view.

3.4.3 Threats to validity

It is impossible to prove that the experiment actually revealed correctly the information it was supposed to extract. In this section, a number of possible factors that might jeopardize the validity of the experiment are presented. Two categories of threats are presented:



(a) Answers to question *post7a*

(b) Answers to question *post7f*

Figure 3.26: Experiment impression

- *The internal validity*. "It is the basic minimum without which any experiment is uninterpretable", meaning that the information collected cannot be used to extract the results
- *The external validity*. "Asks the question of generalizability", meaning that the information collected is correct but cannot be used to extrapolate the results.

Almost all the threats were identified by Campbell[4]. Therefore, in an attempt to avoid excessive references, references to that book will be omitted in the next two sections.

Internal validity

The first problem with one-group pretest-posttest experiment is the effects of *history* ("the specific events occurring between the first and second measurement in addition to the experimental variable"). As mentioned in section 3.2.1, the experiment was conducted without breaks between observation. It is believed that the time difference between the two observations (tests) had no impact on the actual results except that the participants may be a little tired when filling the posttest.

A second rival variable for the selected experiment design is the *maturation*, or the biological changes that happen with participants between observations. Because the TestNForce experiment was short, there is an infinite small chance that any major changes happened that might have impacted the results.

It might be that the participants were confronted with the effects of *testing*. During other experiment, it was observed that the answers to a second test improved for all participants. It could be that participants knew what to expect while doing the posttest but, there was no evidence (major difference in responses) that this happened.

A real threat to the internal validity could have been the experience of the participants with Visual Studio and C#. However, it turned out that the Java background of some participants was more than sufficient to get them up to speed. However, there is no reason to not believe that some participants had an advantage.

The conditions of the experiment were kept as constant as possible. All participants worked in the same environment, got the same treatment - they were all offered water, they all got assistance when needed but, only as much as to unblock them and then communication was neither too formal nor information.

External validity

The first threat to the external validity of the experiment was mentioned by one of the participants. He believes that the assignment was designed in such a way that it highlights the strong points of TestNForce. While such a decision could have a major impact on the experiment, we consider that Jurassic was not selected to put TestNForce in a good light. An important attention was paid to selecting it, as described in section 3.2.3, and the tool was selected for the simple fact that it was suitable for being used with the prototype. Any application can be used, equally well, with TestNForce as long as it is written and C# and it uses `mstest`.

The experience of the participants could jeopardize the external validity of the experiments. However, it is believed that the participants are at least at the level of the average developers in the industry. They proved, by the fast adoption of Visual Studio and by understanding the advantages and disadvantages of TestNForce, that the prior mentioned experience is not a limitation for them.

3.5 Summary

In this chapter, two research questions were answered. This was possible because of the results obtained through the meanings of a user study. Also, it can be concluded that TestNForce is useful and it offers a good user experience.

Another fact observed in this chapter is the quality of the experiment. The participants were satisfied with the quality of the experiment but they agree, together with us, that there is room for improvements.

Chapter 4

Related Work

This related work section presents the existing approaches in establishing the link between the test code and the code under test. The second part gives an overview of the existing support, in Integrated Development Environments (IDE), for establishing this link.

4.1 Traceability

The scientific literature for establishing the link between tests and the code under test can be divided in two categories. Based on the type of the analysis approach there is static analysis [17, 16] and dynamic analysis [10]. Static analysis is usually fast but might lack some information that is available only at runtime; for example, if code is generated on-the-fly (at runtime), then static analysis might ignore it. On the other hand, dynamic analysis can take as long as the program to run, but there is none or little uncertainty about the accuracy of the result[7]. The fact that dynamic analysis is more reliable and much more difficult is also supported by Greevy and Ducasse[13] who found that it is challenging to develop tools which are able to process large volumes of trace data.

The method presented in this paper falls into the dynamic analysis category. However, static analysis methods are presented in this section because the output of those approaches is the same.

The most simple static analysis approach is to exploit the Naming Conventions (NC)[2] like naming schema for test methods; for example: if the tested method is called “Add” then the test should be named “TestAdd” or “AddTest”. Exploiting the test’s names can provide information about which method is covered by a particular test. However, while this method is powerful and able to give good results for those tests which cover one method, it will only identify the method that matches the test name in the case that more than one method is tested. Also, NC based tools’ results can be altered when refactoring legacy code if methods are renamed but tests are not updated; usually this happens when developers are not aware of the relation between unit tests and the code they change[8].

Similar to the previous method, Fixture Element Types (FTE) are presented in [2]. They allow programmers to explicitly mark the units under test. This is an easy and cheap method of traceability that works as long are FTE are properly maintained. Van Rompaey and

Demeyer[17] use Static Call Graphs to determine the methods under test. The results might not be conclusive because they tested it on just three systems, but the precision seems to be under 25%. Their paper compares different static analysis methods and they got to the conclusion that NC is the best choice as long as the naming conventions can be applied.

Many times, helper functions are used in test cases to prepare the environment. A naive approach, would consider that the calls to helper methods are part of the code under test since are called from the unit test. Van Rompaey and Demeyer[17] have come up with a method called Last Call Before Assert (LCBA) that is able to ignore calls to helper methods from the test chain. They assume that the call to the methods under test is the call just before the assert statement. In this way, they ignore any setup calls done in the test prologue and increase the accuracy of the result.

However, LCBA is subject to poor outcomes if developers use assert statement to validate the environment or if the last call before assert is not to the method under test. To overcome the second deficiency, Qusef, Oliveto and De Lucia[16] developed an extension for LCBA based on Data Flow Analysis. Instead of analyzing just the call before assert, they try to identify all statements before assert that affect directly or indirectly the method under test. The authors mention that results could be improved further, for polymorphic contexts, by using dynamic analysis.

All the previous methods are static and try to overcome the fact that runtime information is not available. It is clear that static analysis is a limitation for many. Important research is conducted in this field in order to provide hardware support for dynamic analysis[14] and to improve the existing techniques. However, a major limitation for dynamic analysis is that it requires (partially) compilable code[10].

To the best of our knowledge, the closest approach to our approach has been presented in [10]. The tool developed by Galli et. al. is based on a similar technique but the outcome of the analysis is used, in our case, for establishing the link between tests and code, while in the case of the other paper, test ordering is the goal. Their technique is as follows: before running the tests, the authors instrument the code with trace methods. After running each test individually and aggregating the results, they form groups of tests and try to create a hierarchical relation between them.

4.2 Support in IDEs

IDEs don't provide very much support for tracing unit tests and their associated code. Eclipse Java Development Tools[1] allows developers to navigate between unit tests and code under test. However, this feature is available only if tests are created using a special wizard. Such a constraint reduces the usability of the feature and might prevent the integration with automated build systems.

For Microsoft Visual Studio, the Test Impact Analyzer does a similar job as TestNForce. Jetbrains[2] provides two solutions, ReSharper and dotCover which, combined, can add trace-

[1] http://eclipse.org/jdt

[2] http://www.jetbrains.com

ability features to Visual Studio. dotCover is still in Beta stage and the integration is not complete.

The most common approach for identifying the link between tests and code under test is to do a full test run and analyze the coverage results. This is the approach used by the Visual Studio Test Impact Analyzer and by TestNForce.

Chapter 5

Conclusions and Future Work

In the first chapter, a number of four research questions were proposed. Now, at the end, it is possible to answer them. Below, we present the answer to each of them and give any insightful findings:

1. *Question 1*. Can such a tool be implemented using the currently available tools? As it turned out, such a tool can be implemented using C# and the tools provided by Visual Studio. The implemented solution is not the ideal one and, as described in section 2.3.6, there are better, but more complicated ways of doing it. However, the question was "if" not "how" such a tool can be implemented, therefore we consider this question answered.

2. *Question 2*. Is the tool usable from a performance point of view? During the development process, we encountered a lot of performance issues similar to the ones described in section 5.2. Most of them were overcome and the application performance is above the expected threshold. Some might find TestNForce slow, but there is a certain amount of time under which the analysis time cannot drop because of external factors like the compilation time. Overall, TestNForce responds fast to user queries and spends an acceptable amount of time building the index.

3. *Question 3*. Is the tool useful for developers and testers? Once we considered TestNForce good enough to be used by others, outside the development team, we conducted an user study. A number of developers, that had some test experience, used the tool and expressed their opinion about it. It turned out that TestNForce was mostly useful for the participants and they all used it, with success, to complete the tasks.

4. *Question 4*. Is the tool offering a good user experience? The user study evaluated TestNForce from a usability perspective too. Because of its simple interface, the participants considered the interface intuitive, simple and pleasant. The simplified interface is the result of many hours of brainstorming and sessions of trial-and-error for deciding what can be done without user intervention. The final user interface has just three menu items which we believe is over our best expectations.

5.1 Contributions

We have made the following contributions:

- We have created TestNForce, a tool for establishing the link between tests and code under test. This tool helps developers to identify the tests that must be executed in order to validate code changes.
- We have conducted an user study that revealed developers opinion about TestNForce and the degree to which TestNForce helped them solve programming assignments.

5.2 Memorabilia

This section is dedicated to remarkable events that happened during the development of TestNForce.

1. *One line of code that improved the performance 3 times*. After TestNForce was first run for Jurassic, a "showstopper" (an issue that will prevent TestNForce from being shown to and used by developers) was revealed: building the index took one hour and 45 minutes. This value was over our worst case estimations and it could have been something that proved that implementing TestNForce is something that cannot be done. After about two weeks of digging and debugging, an issue was found in the index builder. The problem was the instead of checking if a method is covered by test methods, the index building was checking the method against all other methods leading to a complexity of $O(n^2)$ instead of $O(n+m)$ where $m << n$. After applying the fix which was simple Linq `Where` predicate, a one line of code fix, the indexing time decreased to 28 minutes. After checking if the results are still correct it became clear that TestNForce was back on track and it is something that can be implemented.

2. *What do Monroe and TestNForce have in common?*. When the development of TestNForce started, we had no idea about all the features that will be included in the final version. Choosing a good name in the beginning is difficult especially when the final goal is not fully defined. Spending time on it would be counterproductive. Therefore, TestNForce was called project codename "Monroe", a temporary name before a more appropriate one can be found. Internally, in the code, there are many references to namespaces starting with "Monroe".

3. *The Pareto principle holds for TestNForce* . The Pareto principle, also known as the 80:20 rule, applies to TestNForce development. Most of the features of TestNForce and the basic "happy path" were created in about 2 months. The rest of the development time (approx. 8 months) was dedicated to adding the rest of the features, handling special cases, tuning the performance and checking the correctness of the results. At least in terms of code, a minimum of 80% of it was written in the two months.

5.3 Future work

Based on the answers given to the questions in the fifth section of the post test and based on the features that were cutoff because of schedule, a number of features will be added to TestNForce in the future.

1. *Navigation from the covering tests window to the actual test code*. This was the biggest pain point for experiment participants. All of them noticed the lack of navigation from the tests list to the test code. Implementing this feature will require a big number of changes because the current index has no information about the files in which the tests reside. There are a number of unanswered problems that will have to be solved before starting the coding phase. One of the most complicated questions is "What happens if the file that holds the test is renamed/moved?". Similar scenarios involved renamed/moved tests and changes to name/location, in general. No matter what the complexity is, the navigation is a number one priority on the "vNext backlog". Adding the feature will give many benefits. In terms of user experience, developers will no longer be disturbed by unusual workarounds. In terms of completion, TestNForce will get closer to what the users expect from such a product.

2. *Console version of TestNForce* . Even though it was not the most desired feature, the console version will give almost endless possibilities for TestNForce. Once the console version will be available, TestNForce will no longer be tied to Visual Studio. Software engineers can then use TestNForce in their build systems, in their own custom checkin gates and even in their own helper scripts. The console version was prototyped, but not released with the current version of TestNForce because of insufficient support of commands. Only a small number of operations are supported by the current console version and there are some stability issues too, stability issues caused by the lack of Visual Studio instance.

3. *Integration with the Visual Studio test platform*. Currently, TestNForce uses `mstest`, but not in the same way as Visual Studio does. Unifying the two will allow developers to run tests only once and update both the index and check if there are any regressions. Moreover, once the integration is completed, TestNForce can take advantage of custom attributes and other reporting features of the VS test platform. To the best of the author's knowledge, the integration is not yet possible, but there are rumors that extension points will be provided in future versions of Visual Studio.

Bibliography

[1] Thomas Ball. The concept of dynamic analysis. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering ESEC/FSE 99*, volume 1687 of *Lecture Notes in Computer Science*, pages 216–234. Springer Berlin, Heidelberg, 1999.

[2] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[3] Bruno Cabral, Paulo Marques, and Luís Silva. Rail: code instrumentation for .net. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 1282–1287, New York, NY, USA, 2005. ACM.

[4] D.T. Campbell and J.C. Stanley. *Experimental and quasi-experimental designs for research*. Rand McNally, 1973.

[5] Y.F. Chen, D.S. Rosenblum, and K.P. Vo. Testtube: A system for selective regression testing. In *Proceedings of the 16th international conference on Software engineering*, pages 211–220. IEEE Computer Society Press, 1994.

[6] A.M. Dean and D. Voss. *Design and analysis of experiments*. Springer texts in statistics. Springer, 1999.

[7] Michael D. Ernst. Static and dynamic analysis: synergy and duality. In *proceeding:996821*, pages 35–35, New York, NY, USA, 2004. ACM.

[8] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[9] M. Fowler. *Patterns of enterprise application architecture*. The Addison-Wesley signature series. Addison-Wesley, 2003.

[10] Markus Galli, Michele Lanza, Oscar Nierstrasz, and Roel Wuyts. Ordering broken unit tests for focused debugging. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 114–123, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design Patterns CD: Elements of Reusable Object-Oriented Software, (CD-ROM)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[12] Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337 – 342, 2004. Special Issue on Software Engineering, Applications, Practices and Tools from the ACM Symposium on Applied Computing 2003.

[13] Orla Greevy and Stephane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. *Software Maintenance and Reengineering, European Conference on*, 0:314–323, 2005.

[14] Markus Mock. Dynamic analysis from the bottom up. In *ICSE Workshop on Dynamic Analysis*, pages 13–17, 2003.

[15] Alan Page and Ken Johnston. *How We Test Software at Microsoft*. Microsoft Press, 2008.

[16] Abdallah Qusef, Rocco Oliveto, and Andrea De Lucia. Data flow based traceability recovery between unit tests and classes under test. In *26th International Conference on Software Maintenance (ICSM), IEEE*, 2010.

[17] Bart Van Rompaey and Serge Demeyer. Establishing traceability links between unit test cases and units under test. *Software Maintenance and Reengineering, European Conference on*, 0:209–218, 2009.

[18] G. Rothermel and M.J. Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering, IEEE Transactions on*, 24(6):401–419, 1998.

[19] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, pages 929–948, 2001.

[20] P.G. Zachary. *Showstopper! The breakneck race to create Windows NT and the next generation at Microsoft*. E-Rights/E-Reads Ltd, 2009.

[21] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production x00026; test code. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 220 –229, april 2008.

Appendix A

Experiment Documents

A.1 Usability survey

Date _____

1. For each of the following selection ranges, specify what should be included in the result.

   Put an 'X' in the cell if you consider it should be included in the result.
   The code model:

   AppWithTests [namespace]
   Program [class]
   Main [method]
   Add [method]
   Subtract [method]

   *Marking an inner node includes all the children.

Selection	[nsp] AppWithTests	[cls] Program	[mtd] Main	[mtd] Add	[mtd] Subtract
1					
6					
8					
10					
15					
10-13					
17					
15-26					
15-24					
8-15					
22					

File: AppWithTests/Program.cs

2. For the second selection method (just declaration selection). Would you expect to get detailed information about the leaf nodes if selecting an inner node or just the aggregated result? What is the reason why you consider this approach more appropriate?

3. What if a class/namespace splits over multiple files? The result should include only the current file or all occurrences?

4. [optional] Do you see another alternative for invoking the action? If yes, please describe it briefly.

Thank you!

A.2 Pretest

Pre-test

In this short survey, a number of questions regarding your experience and attitude towards software development and testing will be asked, in order to get an impression of your skills and expectations. Fill each field with an answer and then save the document. You might encounter words that look like links. Move the mouse cursor over them to get more information.

Candidate ID: Type ID

1. **Please tell us about yourself. This information will not be published; is only needed to put your answers to the next questions in a context.**

1a) What is your age?	Type age here
1b) What is your educational background?	Type education background here
1c) At which university(ies) did you studied?	Type names here, separated by comma
1d) What is your current occupation?	Type occupation here

2. **Please tell us about your software development experience. Rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.**

2a) I consider myself an experienced developer	Choose rating
2b) What is (are) the development environment(s) that you are experienced with	Type here, separate by comma
2c) I consider myself an experienced Visual Studio user	Choose rating
2d) What is (are) the programming language(s) that you are experienced with	Type here, separate by comma
2e) I consider myself experienced with a .NET language	Choose rating
2f) I consider myself an experienced C# programmer	Choose rating
2g) I worked before on large scale software projects	Choose rating
2h) I understand the challenges that arise in software projects	Choose rating
2i) I consider myself familiar with Jurassic	Choose rating

3. **Please tell us about your testing experience. Rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you. For the last question, type a percentage value.**

3a) I consider myself an experienced tester	Choose rating
3b) I write tests for most of the code I am writing	Choose rating
3c) What kind of tests you did in the past?	Type here, separate by comma
3d) I believe that is better to deliver fast, with possible defects than to spend extra time on testing	Choose rating
3e) I believe that one-man projects don't require automated tests	Choose rating
3f) Is it common that the size of the test code to be greater than the size of the tested code	Choose rating
3g) I believe that the amount of resources spent on developing and maintaining test code can be greater than	Choose rating

those spent for the tested code	
3h) How much code coverage is considered "good"?	Type value%

4. Please tell us what you think about the tool described below. Rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.

> 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	Choose rating
4b) I think that such a tool reduces testing time	Choose rating
4c) I think that such a tool reduces the overall development time	Choose rating
4d) I think that such a tool might be annoying	Choose rating
4e) I think that such a tool is solving a real problem	Choose rating
4f) I heard of/used such a tool in the past (could be for any language, not necessarily .NET ones). If the answer is "yes", give some details about it:	
Type answer here	

A.3 Posttest

Post-test

Now you know about TestNForce and what it (cannot) do. In this short survey, a number of questions about the experiment and TestNForce will be asked. Fill each field with an answer and then save the document. You might encounter words that look like links. Move the mouse cursor over them to get more information.

Candidate ID: Type ID

1. **Please tell us about general impression about TestNForce. Rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.**

1a) I like TestNForce	Choose rating
1b) TestNForce is annoying/intrusive	Choose rating
1c) TestNForce is useful	Choose rating

2. **Please tell us about your assignment experience with TestNForce. Rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.**

2a) TestNForce helped me complete the assignment	Choose rating
2b) My test identification result improved with TestNForce	Choose rating
2c) TestNForce makes me more confident when changing unknown code	Choose rating

3. **Please tell us about experience with TestNForce and TFS. Rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.**

3a) The checkin policy for test enforcement is good	Choose rating
3b) The integration between TestNForce and TFS is good	Choose rating
3c) The checkin policy is too restrictive	Choose rating
3d) It should not be possible to bypass the checkin policy	Choose rating

4. **Please tell us about the usability experience of TestNForce. Rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.**

4a) TestNForce is easy to use	Choose rating
4b) TestNForce is slow	Choose rating
4c) The checkin policy for TFS is easy to use	Choose rating
4d) The integration between TestNForce and Visual Studio is good	Choose rating
4e) The messages provided by TestNForce/checkin policy were meaningful and useful	Choose rating
4f) I would prefer TestNForce as a standalone tool	Choose rating

5. **Please tell us to which extent, the following features should be added to TestNForce. Rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they should be added.**

5a) Support for other programming languages	Choose rating
5b) Possibility to exclude certain parts of the project from checking	Choose rating
5c) Incremental update of index	Choose rating
5d) Static code analysis	Choose rating
5e) Integration with the test platform in Visual Studio	Choose rating
5f) More configuration options	Choose rating

6. **Please tell us about your experience with the assignment, during the experiment. Rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.**

6a) The assignment was too hard for me	Choose rating
6b) The time allocated for too short for the assignment	Choose rating
6c) The assignment was interesting	Choose rating
6d) I could have used more guidance	Choose rating
6e) I learned something new	Choose rating

7. **Please tell us about your overall experience during the experiment. Rate each of the statements on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extent they apply to you.**

7a) I enjoyed the experiment	Choose rating
7b) The quality of the assignment was high	Choose rating
7c) The quality of the survey was high	Choose rating
7d) The selected application (Jurassic) was appropriate	Choose rating
7e) The conditions in which the experiment took place were good	Choose rating
7f) My overall impression of the experiment is good	Choose rating

8. **If you have any further remarks, findings, suggestions or other input, now is the time to express them.**
   Type additional comments here

9. **I would like to receive a copy of the experiment's outcome:** Type e-mail address here

A.4 Assignment

Assignment for the new employee

Candidate ID: Type ID

Welcome to Monroe Corporation! Our corporation develops the best software in the industry. Our new and revolutionary product is Jurassic, which is a .NET compiler for JavaScript. We have quite a few developers working on this product but, currently, they are all in Bahamas for a team building event; to be frank, everyone is there now. Without them, we cannot develop new features and no one can make you a proper training – that's why you are actually reading this note instead of having someone presenting the company. Anyway, make yourself confortable and feel like home – there is some water on the table; feel free to take a bottle.

We plan to add some new features in Jurassic but since you are too new here, we can't ask you to start implementing them. However, we would like you to get familiar with the product and test code that we have. While doing that, we would like you to **try to estimate the cost** (in terms of affected functionality) of our new features.

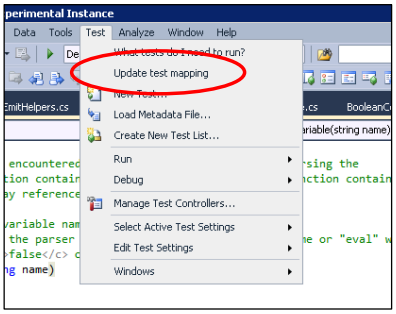
To be frank, there are a few more things that you should know about our process. We hate documentation and that's why we don't write anything; however, we test a lot and prefer to use the tests as demo/documentation for what we do. We have a lot of **tests, 344 to be more precise** and they are quite stable. There are also a few bugs in the product so **only 328 of them pass**.

When we change some code, we run tests to validate the change. Because it would be overwhelming to run all tests each time, we got a tool called **TestNForce** that helps us identify the tests that are impacted by changes. It is quite a simple tool that integrates in Visual Studio 2010.

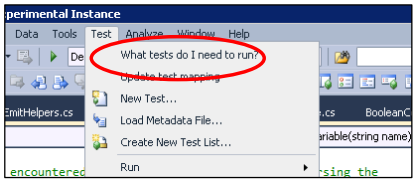
I, the writer of the document, don't know too many about this tool but one of my fellow developers helped me write a short guide. Apparently, there are 4 possible actions:

- Update the mapping index
- See what tests should be run after changing the code
- See what tests cover a specific method
- Checkin the code with TestNForce validation

The test mapping is needed before being able to run TestNForce. It will compile the code, run all tests and create a file with the "tnf" extension, next to solution file (the *.sln file). You don't have to update the test mapping because we did it for you before leaving – we knew that is a time consuming operation, it takes around 25 minutes, and we want you to be as productive as possible. To recreate the mapping, just select the corresponding option from the "Test" menu, in Visual Studio.

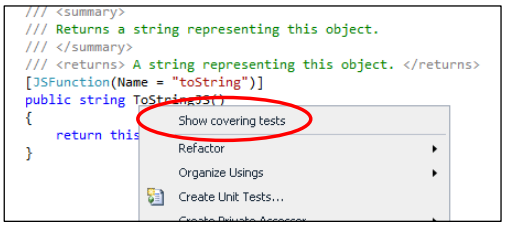


1

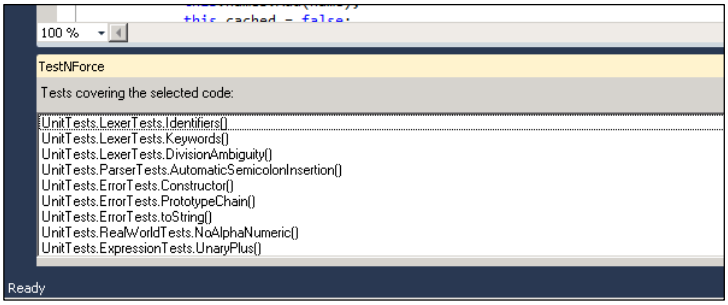


To see what tests need to be run, all you have to do is to invoke the "What tests do I need to run?" option from the Test menu of Visual Studio. This will lead to a list of test that you need to execute in order to validate your code changes. If no changes are done, you will get just a message.

To see what tests cover a specific method, right click the method name and choose "Show covering tests" from the context menu. This will create a list of tests that cover that method or will display just a message if no tests cover the method or the index is not up to date.

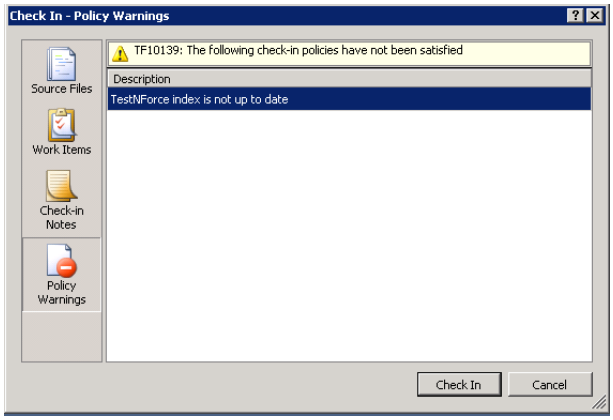


The test mapping window will show either the tests that were affected by your changes or the tests that cover a specific method. If you do more changes after this list is populated, you will need to invoke again the "Show covering tests"/"What tests should I run?" menu item.



To checkin your code, right click the solution node in "Solution explorer" and choose "Checkin". This will pop up a new window and you will be able to submit the code to the source control server. In the Policy warnings section you can check if all the TestNForce policies are satisfied. In order to satisfy all policies, you must include the *.tnf file in your checkin and the file should be up to date (have the modification date greater or equal to the latest changed c# source code file).

2



In case you see this message, update the test mapping file from the Test menu

One more thing: it is your first work days therefore don't spend too much time on this assignment. Go home early. We think that **60 minutes** are enough for the first workday.

Feel free, to ask additional information/assistance if you need at any point of the experiment.

[The rest of this page was intentionally left blank]

3

1. Before you dig into more investigations, we want you to understand why we use **TestNForce.** Look in the file *Jurassic\Jurassic\Compiler\Emit\MethodOptimizationHints.cs* from the Jurassic solution at the function *HasVariable (line 36).* Now try to identify the tests that cover this method without using TestNForce and take notes about your findings. After you are done, use TestNForce to see the tests that cover this method. Compare your findings.
    a. Hints:
        - On your desktop you have "VS2010" which is without TestNForce and "VS2010 TestNForce" which includes TestNForce.
        - Jurassic is located in *<your candidate folder>\jurassic* and you need to open Jurassic.sln in Visual Studio.
        - In case you don't see the solution explorer, try View -> Solution Explorer
        - Don't try to run all tests because it takes a long time and this will not help you solve this problem
        - Try to use "Find references" to see which method covers the method that covers the method that covers…….

2. We want to support new data types in Jurassic. This will require some conversions from the new data types to existing ones. However, we don't know yet if this is a feature with a great risk. Usually we consider great risk those changes that affect many areas, functions, tests, etc. Could you please figure out first what methods we need to change and what tests we would have to run (and maybe fix) if we make this change?
    a. Hints
        - First try to understand what the changes that are required are and how this can be implemented.
        - After you identify the methods, try to get the tests that cover them and decide if this is a great risk change.
        - A good starting point might be the class Jurassic.Compiler.EmitConversion

3. I know that I told you that you will not have to code anything but we changed our mind because as you got here, probably you know more about Jurassic than we do. We have some buggy code in *jurassic\Jurassic\Compiler\Binders\BinderUtilities.cs*; the function that makes trouble is *ResolveOverloads*. It was written by our old developer that left the company 2 years ago and nobody knows what happened to him. Someone tried recently to change the function and did more damage actually; the tests for it fail. Try to fix it and make sure the tests that cover it pass.
    a. Hints:
        - Is more debugging that coding
        - It must be some simple fix; the developer did only minor changes
        - Try to understand what happens there based on comments
        - Use the comments that the old developer left in the function.

4

- Use TestNForce to detect the tests that need to be executed in order to validate your changes

4. Try to checkin your changes to our server.
    a. Hints:
        - You will not be able to do it until you rebuild the test index.
        - Try to understand if the checkin window is user friendly

5. Rebuild the TestNForce index (in case you didn't do it at step 4) so that our next developers will have the latest information about coverage.
    a. Hints:
        - Take a look at the Output window and get an impression about the quality of the information presented.
        - It takes approx. 25 minutes to complete.
        - Don't wait for it to finish. Tell us that you're done. We have more tasks for you.

A.5 Pretest results

Question	1	2	3	4	5	6	7	8
1a) What is your age?	27	26	23	27	26	27	27	25
1b) What is your educational background?	Bachelor Computer Science	Computer science	Advanced Software Engineering	PhD Computer Science	Informatics, MSc Computer Science	Bsc. Computer Science	Science Masters student	Computer Science
1c) At which university(ies) did you studied?	TUDelft, University Leiden	University of Patras, Greece, TU Delft	School of Technology and Managemen t of Leiria (Portugal), University of Leicester (UK)	U. Valladolid, TU Delft	University of Macedonia, TU Delft	North South University, Bangladesh. TUDelft, Holland	Delft University of Technology	Hogeschool Zuyd, TU Delft
1d) What is your current occupation?	Student Master TuDelft	Student Computer Science, Thesis work	M.Sc. Student PhD Student	PhD student	MSc Student, Intern Researcher at SIG	Msc. Student	Self employed, owner of IT consultancy	Student
2a) I consider myself an experienced developer	4	3	4	4	3	3	4	4
2b) What is (are) the development environment(s) that you are experienced with	Eclipse, Borland	Eclipse, Vim	VS and Eclipse	Netbeans, eclipse, VS	Eclipse, VIM	VS, Eclipse	Eclipse, VS	VS, Eclipse
2c) I consider myself an experienced Visual Studio user	2	2	3	3	1.1	4	4	3
2d) What is (are) the programming language(s) that you are experienced with	Java, Haskell, C, C++	C, Java, Python	C# and Java	C, Python, Java	Java, C, Python, R, Stratego	C#, Java, C++	Java, C#	Java, C, C++, C#
2e) I consider myself experienced with a .NET language	1.1	2	4	3	1.1	3	4	4
2f) I consider myself an experienced C# programmer	1.1	2	4	3	1.1	3	4	3
2g) I worked before on large scale software projects	2	2	4	3	2	4	3	2
2h) I understand the challenges that arise in software projects	4	3	4	4	5	4	4	4
2i) I consider myself familiar with Jurassic	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1
3a) I consider myself an experienced tester	2	2	4	3	2	3	3	2
3b) I write tests for most of the code I am writing	1.1	2	3	2	3	4	2	2

3c) What kind of tests you did in the past?	Manual	Unit, Integration	Unit	Unit	Unit testing	Unit, Integration, Acceptace	Unit	
3d) I believe that is better to deliver fast, with possible defects than to spend extra time on testing	3	2	2	3	1.1	2	2	2
3e) I believe that one-man projects don't require automated tests	3	3	2	2	1.1	3	1.1	2
3f) Is it common that the size of the test code to be greater than the size of the tested code	2	3	4	4	4	5	3	3
3g) The amount of resources spent on developing and maintaining test code can be greater than those spent for the tested code	2	2	5	4	3	5	2	3
3h) How much code coverage is considered "good"?	40%	75%	85%	100%	90%	82.50%	70%	90%
4a) I think that I would use such a tool	4	4	5	4	5	3	3	4
4b) I think that such a tool reduces testing time	3	4	5	4	4	4	4	4
4c) I think that such a tool reduces the overall development time	4	4	4	3	3	4	3	4
4d) I think that such a tool might be annoying	3	2	2	4	1.1	4	3	3
4e) I think that such a tool is solving a real problem	3	5	5	5	4	3	3	4
4f) I heard of/used such a tool in the past (could be for any language, not necessarily .NET ones).	No	No	No	No	No	No	No	No

A.6 Posttest results

Question	1	2	3	4	5	6	7	8
1a) I like TestNForce	4	4	5	4	5	4	4	4
1b) TestNForce is annoying/intrusive	2	1	2	3	1	3	3	2
1c) TestNForce is useful	4	4	5	5	5	4	4	5
2a) TestNForce helped me complete the assignment	4	4	5	5	5	4	5	4
2b) My test identification result improved with TestNForce	5	5	5	5	5	4	5	3
2c) TestNForce makes me more confident when changing unknown	3	5	4	3	5	4	4	4
3a) The checkin policy for test enforcement is good	2	4	3	3	5	3	3	4
3b) The integration between TestNForce and TFS is good	3	5	3	3	5	3	3	4
3c) The checkin policy is too restrictive	2	2	3	3	1	4	4	3
3d) It should not be possible to bypass the checkin policy	2	4	3	3	5	2	2	3
4a) TestNForce is easy to use	4	5	5	4	3	5	4	5
4b) TestNForce is slow	3	3	2	3	4	3	3	2
4c) The checkin policy for TFS is easy to use	2	4	3	3	5	4	4	2
4d) The integration between TestNForce and Visual Studio is good	4	5	4	3	3	5	4	4
4e) The messages provided by TestNForce/checkin policy were	3	5	3	4	5	3	4	4
4f) I would prefer TestNForce as a standalone tool	2	3	1	3	1	2	2	2
5a) Support for other programming languages	5	4	3	4	5	4	4	5
5b) Possibility to exclude certain parts of the project from checking	4	3	3	3	3	4	2	5
5c) Incremental update of index	3	3	5	5	5	5	5	4
5d) Static code analysis	2	4	3	3	5	4	2	4
5e) Integration with the test platform in Visual Studio	4	4	5	5	3	4	4	5
5f) More configuration options	2	2	3	3	3	5	3	4
6a) The assignment was too hard for me	2	3	2	1	1	3	2	2
6b) The time allocated for too short for the assignment	2	2	1	3	1	3	2	2
6c) The assignment was interesting	3	5	4	5	4	4	4	5
6d) I could have used more guidance	2	2	2	3	1	3	1	3
6e) I learned something new	4	5	4	3	5	5	3	5
7a) I enjoyed the experiment	3	4	4	4	5	4	4	4
7b) The quality of the assignment was high	4	5	4	4	4	3	4	5
7c) The quality of the survey was high	3	4	4	3	5	4	4	4
7d) The selected application (Jurassic) was appropriate	3	3	5	5	2	3	4	4
7e) The conditions in which the experiment took place were good	4	4	4	3	5	4	4	4
7f) My overall impression of the experiment is good	4	5	5	3	5	4	4	4

Appendix B

Glossary

DTO: Data Transfer Object

Memorabilia: Matters or events worthy to be remembered; points worthy of note

TFS: Team Foundation Server

TNF: TestNForce

vNext: Next/future version of a software product

VS: Visual Studio