This is a preprint of an article:
Lech Madeyski and Marcin Kawalerowicz, "Software Engineering Needs Agile Experimentation: A New Practice and Supporting Tool", in *Software Engineering: Challenges and Solutions* (L. Madeyski, M. Śmiałek, B. Hnatkowska, and Z. Huzar, eds.), vol. 504 of *Advances in Intelligent Systems and Computing*, pp. 149–162, Springer, 2017. DOI: 10.1007/978-3-319-43606-7_11
The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-43606-7_11
[BibTeX] Draft: http://madeyski.e-informatyka.pl/download/MadeyskiKawalerowicz17.pdf

Software Engineering Needs Agile Experimentation: A New Practice and Supporting Tool

Lech Madeyski[1] and Marcin Kawalerowicz[2]

[1] Wroclaw University of Science and Technology, Faculty of Computer Science and Management, Wyb.Wyspianskiego 27, 50-370 Wroclaw, POLAND,
Lech.Madeyski@pwr.edu.pl

[2] Opole University of Technology, Faculty of Electrical Engineering, Automatic Control and Informatics, ul. Sosnkowskiego 31, 45-272 Opole, POLAND,
m.kawalerowicz@po.opole.pl

Abstract. This article proposes a novel software engineering practice called Agile Experimentation. It aims mostly small experiments in a business driven software engineering environment where a developer is a scarce resource and the impact of the experimentation on the return-of-investment driven software project needs to be minimal. In such environment the tools used for the sake of research need to have virtually no negative impact on the developers, but simultaneously those tools need to collect high quality data to perform sound enough quantitative analyses. In order to fulfill those requirements, and to support the Agile Experimentation practice, we co-developed a tool called NActivitySensor that gathers the data about the developers activities in a widely used Integrated Development Environment—Visual Studio. The proposed Agile Experimentation practice and the developed tool complement each other to support lightweight experimentation in real-world software development settings.

Keywords: agile experimentation, empirical software engineering, experimentation in software engineering, NActivitySensor

1 Introduction

Performing experiments in industrial software development environment is difficult and quite expensive [8]. It is usually hard to replicate the software engineering experiments [1]. The same problem is magnified in Software Engineering experiments conducted in industrial environment. There is usually no budget for conducting the experiment twice, with a classic method (used to date) as well as using the new approach (the one being evaluated). It takes a rich country and research institute to perform controlled replication experiment in real software project with professional developers [23]. It seems like in last years the software engineering scientists realized that there is a need for experimentation. In fact,

nowadays it is more and more difficult to publish research papers that do not include an empirical evaluation of new methods, practices, technologies or tools.

While we notice the research activities concerning experimentation in software engineering of the large multinational corporations like IBM (http://research.ibm.com/ - 213 publications in "Programming Languages and Software Engineering" area), Microsoft (http://research.microsoft.com/ - 203 research projects in "Software development, programming principles, tools, and languages" area) or Google (http://research.google.com/ - 177 publications in "Software Systems" area)[3], we do see the lack of research motivation in smaller companies all around the world. Running real world software engineering project is a business with many interested parties, the most important of which is the customer. Unfortunately, the main priority for a customer is the return of investment (ROI), not the experimentation with a new software engineering method. Despite it could be beneficial in later projects. Customers fear that it will cost too much and there will be too much hassle while performing the experiments. For example developers will be pulled away from their actual job and software development tasks they should focus on. The developers themselves fear they will have more responsibilities in they day-to-day work. To overcome such difficulties and to fill the existing gap with regard to lack of software engineering experimentation practices and supporting tools, we call for something we denominated Agile Experimentation (more after simple definition from Merriam-Webster's Learner's Dictionary: "quick, smart, and clever" than after Agile Software Development). It is a way to perform empirical studies and research in real-world software projects with no or minimal impact on people involved in those projects and their schedules. But in the same time what we propose should give researchers the way to perform effective experimentation to reach reliable enough conclusions on a subject being investigated (e.g., the impact of a new practice, method or tool).

In this paper, we start from our motivation, presented in Section 2, on which we build our agile experimentation "manifesto" formulated in Section 3, where the idea is thoroughly explained. Then we focus on a tool that we co-developed to aid the agile experimentation. It is called NActivitySensor after its predecessor called Activity Sensor [18,20,19] that was developed under the supervision of one of the authors of this paper. The new tool was built for the Microsoft .NET Framework development, while its predecessor was built for Java. We describe the new tool in detail in Section 4 and in Appendix A. Discussion and future work can be found in Section 5.

2 Motivation

The reasons given by scientists and professionals for not experimenting in software engineering are greatly outlined in the article "Should computer scientists experiment more?" by Tichy [22]. Among them are:

[3] The numbers gathered in May 2016.

- Traditional scientific method is not applicable.
- The current level of experimentation is good enough.
- Experiments cost too much.
- Demonstration till suffice.
- There is too much noise in the way.
- Progress will slow.
- Technology changes too fast.
- You will never get published.

Tichy fights those excuses one by one in his paper with argumentation that will most likely resonate with average scientist but not a business person. It is because the researcher is most likely to aim at the scientific excellence and businessman is most likely to be interested in ROI. The argument that ”experiments cost too much” is the hardest to fight in a ROI-oriented environment such as industrial software development. It is a fact that performing an experiment costs. It might be beneficial in a long term, but in terms of immediate ROI, it is always burdened with an additional cost. It is the reason why so few smaller companies are willing to take the additional experimentation costs into consideration while racing for the customer on highly competitive market. For the same reason there are so few industrial professionals available for experimentation. They are considered to be valuable resource to waste their time for additional tasks like taking part in experiments. The professionals themselves seem to be rather ambivalent to the idea of taking part in experiments. From one point of view, they know that the results are potentially beneficial but from the other, they fear they will have less time for their duties towards the employer.

Thomke in his article ”Enlightened Experimentation - The New Imperative for Innovation” names the cost of the experimentation the main damper for the companies to create great products [21]. His list of ”Essentials for Enlightened Experimentation” gives four rules for companies to be more innovative:

1. Organize for rapid experimentation.
2. Fail early and often, but avoid mistakes.
3. Anticipate and exploit early information.
4. Combine new and traditional technologies.

3 Agile Experimentation ”Manifesto”

We think it is a time to give both the business professionals and researchers ready to use tool-set to align their goals in software engineering. Researchers want to perform controlled experiments that will give reliable enough conclusions leading to improvements in software engineering. Business professionals (e.g., an engineer working on a project or manager supervising it) are mostly interested in return of investment. The work done should make the customer happy and bring benefit for the organization. The results of potential experimentation is interesting for them only if the main goal is met and the additional experimentation cost is acceptably low. We want to give the methods and tools they can use to

better meet those requirements. We gathered those methods and tools in popular nowadays form of "manifesto" [2,3,6,7]. We have played with the form and proposed Agile Experimentation "Manifesto". The target of our "manifesto" are both the researcher as well as the practitioner willing to perform experiments. Some of the items might seem obvious for an experienced researcher but they are not necessary so obvious for an average practitioner. Thus the abstraction level of the items. Our "manifesto" contains the following rules:

1. Use small-n and single case experiments rather than large scale experiments to cut costs and enable experimentation.
2. Care about the power of your experiments to reduce waste.
3. Search for the best experiment design that fits your settings.
4. Use friction free tools for data gathering to not interfere with the real-world development environment.
5. Use just-in-time quantitative data rather than late, post project qualitative surveys to enable early informed decisions (on a basis of quantitative data instead of late anecdotal information).

What really hides behind those statements? How are they important for a business professional or a researcher wanting to do experiment in a real-world software engineering project.

3.1 Small-n and single case experiments

As we mentioned earlier in this paper the main driver of a real-live business software engineering project is ROI, which is what the customer is most interested in. He puts his money into the project and expects to get the best possible software in exchange. There is no place for large scale experimentation in such project. There is no money for an experiment with a large number of professional developers. In fact there can be a problem with finding large number of professional developers willing to sacrifice their precious time for any kind of experiment. What can be done to change this situation? There is a way used by pharmacists and medical scientists for years. It is called small-n and single case experimentation [5,10]. Those are special kind of experiments designed especially for very small samples (small-n) or even one participant (single-case). They are very helpful in clinical trails where the researcher do not have a large set of patients or when you study human behaviour. Often such kind of experiments are done as a low cost pre-studies before large scale and more expensive experiments.

This method took inroads into the software engineering already [24]. The reasoning is simple. Software development team is usually a small group of people (small-n) working on a single software project. In fact every developer working in a project is a case for itself. If psychologists are using small-n and single case experiments to study human behaviour why not use it in software engineering?

We are using this approach to study our extension to TDD (Test Driven Development) that we called CTDD (Continuous Test Driven Development) [17] in

a real, industrial software development environment. We have promising results using the agile experimentation to study TDD vs. CTDD. In fact the idea for agile experimentation was actively trialed during this experiment. We have used our own principles to design and conduct the TDD vs. CTDD experiment. We are intending to publish a paper discussing the results and our experience with agile experimentation as soon as we finish the ongoing experiment.

The idea behind our experiment it to extend the commonly used TDD chain of actions:

write the test → *execute it* → *see if it fails* → *satisfy the test* → *run the test* → *see if it succeeds* → *refactor* → *run the test* → *see if it still succeeds.*

The extension involves using continuous testing (unit tests performed on a background thread in the development environment). Thanks to that technique we can eliminate the need of manually starting the tests and thus improve software development. This kind of improvement can be seen as a modification of the existing TDD practice in contrast to combining different software development practices together (e.g., the TDD and pair programming practices [13,14,15]). In CTDD the chain of actions is shorter:

write the test → *see if it fails*→ *satisfy the test* → *see if it succeeds* → *refactor* → *see if it still succeeds.*

The reasoning is that a small increase of productivity in an individual developer can make a big impact in the whole developers community. The question is how omitting the need of manually executing the tests will impact the overall performance of the developer. That is what we want to estimate by conducting research following the idea of agile experimentation. We had to our disposal only one small (measured in terms of developers) project where only two developers were contracted. An ideal small-n experiment with two conditions (condition A: TDD and condition B: CTDD). But doing small-n and single case experimentation is challenging in another aspect. Precisely because of the small number of participants the researcher needs to be extra caution to be able to achieve enough statistical power to detect the difference between two conditions (software development practices, TDD and CTDD, in this particular case).

3.2 Care about power of your experiments

The so called A/B experiments [12] are quite popular in the mainstream web development lately. They are used to perform simple online test where a new version of an existing website is created with, e.g., slightly different layout, colours, button orientation or alike and taken online alongside the old version. The old version (A - baseline) is showed to a randomly assigned part of the online audience and the new version (B - with intervention) to the other. The visitors behaviour is then recorded. Is the new layout causing them to stay on the page longer? Are they "liking" the page more if the colours are different? Do they click the "order" button more often in the new version? What is done here is a randomized controlled experiment, using between-subject design, where two groups of subjects are simultaneously tested: one being the control group (A) and one being under "treatment" (B). The subjects are simultaneously tested

but assigned randomly to one of the groups, A or B. What is important is the ability to detect statistically significant difference between the treatments, i.e., the statistical power of the experiment is crucial. The power of any test of statistical significance is formally defined as the probability that a false null hypothesis will be rejected if there is no difference between the treatment and the baseline.

Researchers should be interested if the recorded difference between A and B is in fact statistically significant and, last but not least, what is the size of the effect [15,16,11]. The power of the test is especially important if performing small-n and single case experiments, see Section 3.1. To detect an effect of reasonable size the person performing agile experimentation will have to analyze power taking into consideration the small sample size. Another important aspect is generalization from such experiments. Agile experimentation may impede generalisation, but the threads to external validity can be minimized by the fact that agile experiments are conducted in a real world environment, by real software developers, not in a laboratory environment by inexperienced student subjects as is often the case.

In a simple online A/B test the, so called, random allocation is used. A visitor is randomly put into a group that sees A or B version of a web site. It is enough for such a simple experiment, but what if a researcher performs a more sophisticated small-n or single case experiment? In such case, the more possible experiment arrangements the better. The researcher needs to eliminate all competing explanations for the effect he is observing.

Let us consider an experiment with four phases. Every phase is either a baseline (A) or a treatment condition (B). The moment when to switch between A and B can be randomly chosen. This way the possible arrangements are: switch after the first phase - ABBB, switch after the second phase - AABB, or after the third phase - AAAB. The set of possible arrangements is quite small. What if the researcher decides to use a completely random approach? This means to simply choose every phase randomly? Using this approach he will get 16 (2^4) possible arrangements like this: AAAA, BAAA, ABAA, AABA, AAAB, BBAA, ABBA, AABB and so on. But some of the arrangements are not desirable from the beginning (like AAAA or BBBB). In that case the researcher can decide to gather the phases in blocks. For example always in pairs like AABBAA, AAAABB, BBAAAA, and so on.

Let us consider the TDD vs. CTDD experiment. Lets assume we have 8 modules to do. We have 2 software developers so each one may get 4 modules to write. There are 6 possible arrangements within these 4 modules (4 observations, 2 for A and 2 for B): AABB, ABBA, BBAA, BAAB, ABAB, BABA. For 2 developers there is $6 * 6 = 36$ possible arrangements. Maybe switching from the module/package level to the class level (the number of classes is usually larger than the number of packages in Java or solutions in .NET) in the TDD vs. CTDD experiment will be give us higher power to detect the difference? Maybe another design of experiment will be better (for example randomized block design)? This kind of a priori consideration can pay off if the researcher strives for reliable results.

3.3 Searching best experiment design

Finding an appropriate design for a given experiment in software engineering is not an easy task. If it is an small-n or single case experiment it might be even harder because it is not common in software engineering. There might be a need to extrapolate the types of experiments from different fields of study onto software engineering. For example, Dugard et al. [4] give a set of example experimental designs. There is a lot to choose from: single-case/small-n one way, single-case randomize block design, small-n repeated measures, small-n repeated measures with replicates, two-way factorial single-case/small-n, single-case AB, single-case ABA, multiple baseline AB, multiple baseline ABA. One design may be better for one kind of experiment and the other for another type. But how to choose the right one? The one that will answer the posed research question in the best way? There is not an easy way to choose a design. There is no tool for searching the available design given project and research constrains. For example in our research about TDD vs. CTDD we had a small project that was contracted for 2 developers, having 160 hours/month in the period of at least one year, with estimated 37 modules (400 classes) to write. We searched the possible set of designs answering the following important questions [4]:

- Do we have at least two participants? Yes, we have.
- Do we have two conditions to compare? Yes.
- Will each participant receive each of the condition on at least two occasions? Also yes (we have a lot of modules/classes to write and if we consider one as an "occasion" then we are good to go).
- Is it possible to randomly assign conditions to each participant? Yes - we can write a software tool that will randomly assign a class to a given treatment.

"A small-n repeated measure design with replicates" experimental design seemed to be the most suitable for our needs.

3.4 Use friction free tools for data gathering

Software developers refer to tools that do not need much attention while in use as "friction free tools". It is because "friction free tools" are not generating "resistance" while in use. Meaning they need no special attention from the developer. It is quite important to use such tools in agile experimentation while gathering data. Agile experimentation tools should work without the developer attention. They have to integrate seamlessly with the developer environment. After simple installation and minimal configuration they should be up and running. Such friction free data gathering software tools need to be quite resilient. If the data gathering relies on network communication and the developer decides to work from home, they have to provide fall back scenarios so the data will not be lost.

What kind of tools will be needed in any given research depends on the planned experiment. There is quite a number of tools already written and available for various range of experiments. That was the case in our empirical evaluation of new approaches to software engineering - TDD vs. CTDD. We found

very good continuous testing plug-in for IDE that was used in the project under investigation. We needed only to extend it with the data gathering capabilities. It was an open source project so we "forked it", meaning created our branch of source code and added a small data gathering part. We saved measurements like: when, what test and with what outcome were run, how long it took to complete them. The detailed description of the tool we called AutoTest.NET4CTDD can be found in [17].

The second tool we created was a small randomized block generation. We mentioned it in 3.2. It was a tool that randomly assign a condition to a given class. It simply assigned a newly created class to TDD or CTDD group by adding a specially formatted line at the beginning of the file. The line was read by AutoTest.NET4CTDD and the plug-in acted accordingly. It automatically turned on or off the continuous testing.

We needed one more tool: reliable event recorder for our IDE. We were quite surprised when we realized there is none available. So in this case we have written it from scratch and open sourced it for the community. The NActivitySensor tool is described in Section 4.

3.5 Use just-in-time quantitative data

We do not advocate discontinuing the traditional research approaches involving semi-quantitative or qualitative data gathered after the experiment using various kinds of surveys. We suggest to enrich this kind of data with a quantitative data based on experiments, both, large and small scale (e.g., small-n, single-case). We strongly believe that such data can be gathered during the research without an additional effort from the researcher and the subject. This data gathering can occur in a just-in-time manner. Meaning the data is gathered in real-time as it happens and is logged immediately for researcher to be used in his convenience. We have build our TDD vs. CTDD research tools in that way. Both AutoTest.NET4CTDD[4] and NActivitySensor[5] are storing the data in a central database from where they can be used as they are needed.

4 NActivitySensor

For the purpose of this agile experimentation we created a tool called NActivitySensor. It is a "friction free" kind of software tool we described in Section 3.4. It is designed to gather the real-time quantitative data while working. NActivitySensor was developed with the third and forth Agile Experimentation rule in mind. It was co-developed at the software development company one of the authors is running. The company is a mostly Microsoft .NET shop that uses C# as his primary language. The main IDE for .NET is Microsoft Visual Studio (VS). Investigation of the VS extensions showed no tool that can be used to record

[4] Available at https://github.com/ImpressiveCode/ic-AutoTest.NET4CTDD

[5] Available at https://github.com/ImpressiveCode/ic-NActivitySensor

the developer actions in IDE. The data recorded by such tool could then be used in different type of agile experimentation. One of the authors of this paper supervised the creation of similar tool that integrated with Java IDE Eclipse. This tool was called Activity Sensor and developed back in 2006 at Wrocław University of Technology. We decided to name the new tool NActivitySensor. N in the name is often used in the .NET tools and libraries as the indication of the, in most cases, Java based ancestors. NActivitySensor integrates with VS IDE and hooks to Development Tools Environment (DTE) and its ”Test subsystem”.

We are currently using the NActivitySensor to gather data for our research on Continuous Test Driven Development (CTDD)[17]. The detailed description of NActivitySensor is given in Appendix A.

5 Discussion and future work

The proposed agile experimentation practice and supporting tool are proposed to make experimentation in real-world, industrial settings more widely adapted. The agile experimentation practice and supporting tool set will be refined as part of our further research. We are currently working on a new publication where we are discussing the TDD vs. CTDD experimentation. This research is done solely according to the agile experimentation principles. The overall experiences regarding agile experimentation from this ongoing project are promising. We were able to harness the most of our ”manifesto” principles to action. In this research we are targeting professional developers in business driven projects. It is not easy to get the permission to experiment in such environment. Thus if permission is granted it is advisable (to say the least) to make the most of it. By using well known small-case and single-n experiments designs (1) and to take care about the power of the experiment up front (2). The act of experimenting should not impact the work of the developers much (4). With this goal in mind the NActivitySensor was developed. It gathers the information about developer activities in the Visual Studio IDE without disturbing developer's work. The data is then gathered immediately for the sake of quantitative research. We will work further on the agile experimentation idea in order to provide a tool for searching the best experiment design for a given project and research constraints (3).

Appendix A - NActivitySensor

This appendix contains description of the details of NActivitySensor Visual Studio Add-in. NActivitySensor is available as an extension [9] at Visual Studio Gallery. Visual Studio Gallery is a tools, controls, and templates distribution platform used in Visual Studio. NActivitySensor is maintained and supported as a free extension by its creators. It can be installed from withing Visual Studio by using ”Extension and Updates...” from the ”Tools” menu (see Figure 1). After successful installation NActivitySensor writes the activities log to a

new output window with the same name (see Figure 2). It is possible to configure the extension to write the activity log into a database. The database engine supported in NActivitySensor is Microsoft SQL Server. The database connection string is set in NActivitySensor.dll.config in the extension installation directory (which is `C:\Users\Account_Name\AppData\Local\Microsoft\VisualStudio14.0\Extensions`). The key for the connection string is stored in NActivitySensor.MSSql.ConnectionString. Also it is possible to configure the extension on a project level. In order to do it a configuration file in the Visual Studio solution directory is needed. It needs to be called NActivitySensor.config. Example configuration file is showed on Listing 1.

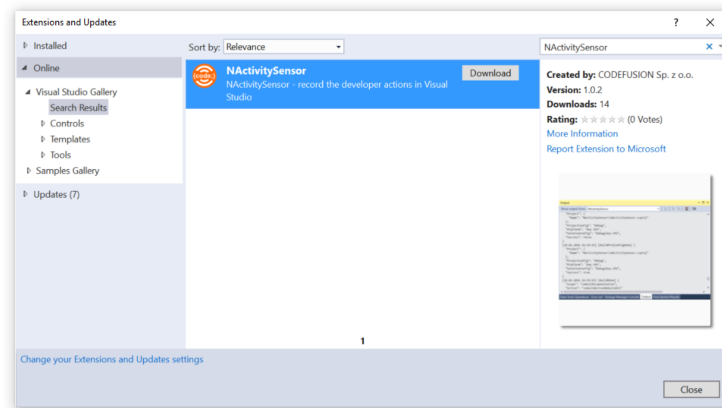

Fig. 1. NActivitySensor installation in Visual studio 2015

Listing 1. NActivitySensor.config file

```
<?xml version="1.0"?>
<configuration>
  <appSettings>
    <add key="NActivitySensor.MSSql.ConnectionString"
          value="Data␣Source=192.168.1.102;
Initial␣Catalog=ExCalcNActivitySensorReports;
User␣Id=NActivitySensor;
Password=NActivitySensor;
MultipleActiveResultSets=True"/>
  </appSettings>
</configuration>
```

The internal storage format for the activities is JSON. JSON stands for (JavaScript Object Notation) and is a widely used as a data exchange format. It was used because it is both: (1) easy to read and write by a human and a machine and (2) its structure does not have to be defined beforehand. We are using JSON

to store various activities with varied format. For example Listing 2 shows the log for the activity 'DocumentOpened' (opening a document in IDE) and Listing 3 shows the 'BuildBegin' (starting of project building in Visual Studio).

Listing 2. DocumentOpened activity example log

```
[22.03.2016 15:24:21] [DocumentOpened] {
  "Name": "Examples.cs",
  "Kind": "{8E7B96A8–E33D−11D0–A6D5−00C04FB67F6A}",
  "Path": "C:\\Dev\\ClaToDot\\ClaToDot.Demo\\" }
```

Listing 3. BuildBegin activity example log

```
[22.03.2016 15:28:27] [BuildBegin] {
  "Scope": "vsBuildScopeSolution",
  "Action": "vsBuildActionRebuildAll" }
```

Table 1 shows all hooked-up events recorded by the NActivitySensor.

Table 1. Microsoft Visual Studio events hooked in NActivitySensor

Event group	**Event**	**Event group**	**Event**
SolutionEvent	SolutionOpened, SolutionBeforeClosing, SolutionRenamed, SolutionQueryClose, SolutionProjectRenamed, SolutionProjectRemoved, SolutionProjectAdded,	TextEditorEvent	LineChanged
BuildEvent	BuildDone, BuildProjConfigDone, BuildBegin, BuildProjConfigBegin	TaskEvent	TaskRemoved, TaskNavigated, TaskModified, TaskAdded
UserEvent	UserInactive, UserActiveAgain	FileItemEvent	FileItemRenamed, FileItemRemoved, FileItemAdded
PluginEvent	Connect, Connection, Disconnection, AddInsUpdate, StartupComplete, BeginShutdown	FindEvent	FindDone
WindowEvent	WindowMoved, WindowCreated, WindowClosing, WindowActivated, WindowPaneUpdated, WindowPaneClearing, WindowPaneAdded	DebuggerEvent	DebuggerExceptionThrown, DebuggerException-NotHandled, DebuggerEnterRunMode, DebuggerEnterDesignMode, DebuggerEnterBreakMode, DebuggerContextChanged
SelectionEvent	SelectionChange	CommandEvent	CommandBeforeExecute, CommandAfterExecute
		DocumentEvent	DocumentClosing, DocumentSaved, DocumentOpened

The data is stored in the Microsoft SQL Server database. Additionally the data is echoed back into VS output windows as a fall-back for non functioning database or network (it is possible to record the data from the output window to a file).

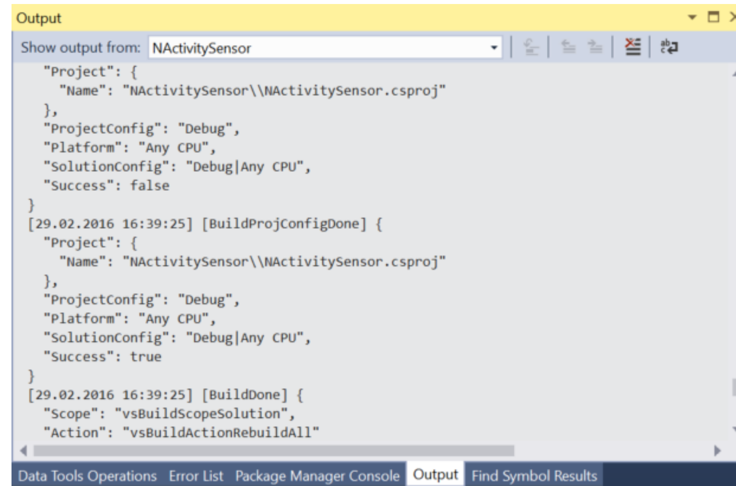

Fig. 2. NActivitySensor output window in Visual studio 2015 IDE

References

1. Basili, V.R.: What's so hard about replication of software engineering experiments? https://www.cs.umd.edu/~basili/presentations/RESER%20Keynote.pdf (October 2011), accessed: 2016-03-18
2. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D.: Manifesto for Agile Software Development (2001)
3. Bonér, J., Farley, D., Kuhn, R., Thompson, M.: The reactive manifesto. http://www.reactivemanifesto.org/ (2014), accessed: 2016-03-29
4. Dugard, P., File, P., Todman, J.: Single-case and Small-n Experimental Designs: A Practical Guide to Randomization Tests, Second Edition. Routledge (2012)
5. Gast, D., Ledford, J.: Single Case Research Methodology: Applications in Special Education and Behavioral Sciences. Taylor & Francis (2014)
6. Guevara, P.C.: Manifesto for minimalist software engineers. http://minifesto.org/ (2013), accessed: 2016-03-29
7. Harman, M., Jia, Y., Langdon, W.B.: A Manifesto for Higher Order Mutation Testing. In: Proceedings of the Third International Conference on Software Testing, Verification, and Validation Workshops. pp. 80–89. ICSTW '10, IEEE (2010)
8. Juristo, N., Moreno, A.M.: Basics of Software Engineering Experimentation. Springer Publishing Company, Incorporated, 1st edn. (2010)
9. Kawalerowicz, M., CODEFUSION: Microsoft Visual Studio Extension NActivitySensor. https://visualstudiogallery.msdn.microsoft.com/4675d6fb-2608-48ed-ae0a-320b3a756047 (2013-2016), accessed: 2016-03-18
10. Kazdin, A.E.: Single-case Research Designs: Methods for Clinical and Applied Settings. Oxford University Press (2011)
11. Kitchenham, B.A., Madeyski, L., Budgen, D., Keung, J., Brereton, P., Charters, S., Gibbs, S., Pohthong, A.: Robust Statistical Methods for Empirical Software

Engineering. Empirical Software Engineering (in press) (2016), http://dx.doi.org/10.1007/s10664-016-9437-5, DOI: 10.1007/s10664-016-9437-5
12. Kohavi, R., Longbotham, R., Sommerfield, D., Henne, R.M.: Controlled experiments on the web: survey and practical guide. Data Mining and Knowledge Discovery 18(1), 140–181 (2008), http://dx.doi.org/10.1007/s10618-008-0114-1
13. Madeyski, L.: On the effects of pair programming on thoroughness and fault-finding effectiveness of unit tests. In: Münch, J., Abrahamsson, P. (eds.) Product-Focused Software Process Improvement, Lecture Notes in Computer Science, vol. 4589, pp. 207–221. Springer Berlin Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-73460-4_20, DOI: 10.1007/978-3-540-73460-4_20
14. Madeyski, L.: Impact of pair programming on thoroughness and fault detection effectiveness of unit test suites. Software Process: Improvement and Practice 13(3), 281–295 (2008), http://dx.doi.org/10.1002/spip.382, DOI: 10.1002/spip.382
15. Madeyski, L.: Test-Driven Development: An Empirical Evaluation of Agile Practice. Springer, (Heidelberg, London, New York) (2010), http://dx.doi.org/10.1007/978-3-642-04288-1, DOI: 10.1007/978-3-642-04288-1
16. Madeyski, L., Jureczko, M.: Which Process Metrics Can Significantly Improve Defect Prediction Models? An Empirical Study. Software Quality Journal 23(3), 393–422 (2015), http://dx.doi.org/10.1007/s11219-014-9241-7, DOI: 10.1007/s11219-014-9241-7
17. Madeyski, L., Kawalerowicz, M.: Continuous test-driven development—a novel agile software development practice and supporting tool. In: Maciaszek, L., Filipe, J. (eds.) ENASE 2013 - Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering. pp. 260–267 (2013), http://madeyski.e-informatyka.pl/download/Madeyski13ENASE.pdf, DOI: 10.5220/0004587202600267
18. Madeyski, L., Piechowiak, A.: Exclipse plug-in activity sensor. http://sens.e-informatyka.pl/projekty/activity-sensor/ (2006), accessed: 2016-03-18
19. Madeyski, L., Szała, Ł.: Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study. IET Software 1(5), 180–187 (2007), http://dx.doi.org/10.1049/iet-sen:20060071, DOI: 10.1049/iet-sen:20060071
20. Madeyski, L., Szała, Ł.: The Impact of Test-Driven Development on Software Development Productivity — An Empirical Study. In: Abrahamsson, P., Baddoo, N., Margaria, T., Messnarz, R. (eds.) Software Process Improvement, Lecture Notes in Computer Science, vol. 4764, pp. 200–211. Springer Berlin Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-75381-0_18, DOI: 10.1007/978-3-540-75381-0_18
21. Thomke, S.: Enlightened Experimentation - The New Imperative for Innovation. Harvard Business Review 79(2), 66–75 (2001)
22. Tichy, W.F.: Should computer scientists experiment more? Computer 31(5), 32–40 (May 1998)
23. Vokáč, M., Tichy, W., Sjøberg, D.I.K., Arisholm, E., Aldrin, M.: A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment. Empirical Software Engineering 9(3), 149–195 (Sep 2004)
24. Zendler, A., Horn, E., Schwärtzel, H., Plödereder, E.: Demonstrating the usage of single-case designs in experimental software engineering. Information & Software Technology 43(12), 681–691 (2001), http://dx.doi.org/10.1016/S0950-5849(01)00177-X, DOI: 10.1016/S0950-5849(01)00177-X