Software Engineers And Nuclear Engineers: Teaming Up To Do Testing

Diane Kelly[2], Nancy Cote[1], Terry Shepard[2]
[1]*Department of National Defense (Canada),* [2]*Royal Military College of Canada,*
cote.mmn@forces.gc.ca, Kelly-d@rmc.ca, shepard@rmc.ca

Abstract
The software engineering community has traditionally paid little attention to the specific needs of engineers and scientists who develop their own software. Recently there has been increased recognition that specific software engineering techniques need to be found for this group of developers. In this case study, a software engineering group teamed with a nuclear engineering group to develop a software testing strategy. This work examines the types of testing that proved to be useful and examines what each discipline brings to the table to improve the quality of the software product.

1. Introduction

There is a long history of software being developed by scientists and engineers who are not primarily computing professionals. As noted by Segal [17], these people are comfortable with and often very skilled at programming. However, their primary focus is the deliverable related to their application discipline – they are interested in what the software can do for them. Their strengths are the models they develop, the models that are ultimately rendered into software. They are the "caretakers" of the models. However, a caretaker of the code is needed, and this is where the software engineer can add to the quality of the software product.

The case study described in this paper involves the interaction between the software engineering research group (SEG) and the nuclear engineering research group (NE) at the Royal Military College of Canada. The immediate purpose of the interaction was to select useful and usable testing steps that would enhance the quality of a piece of software written by one of the members of the NE. The longer-term purpose was to learn more about effective software engineering techniques that could become part of the arsenal of the members of the NE and other scientific developers. An important aspect of the particular piece of software used was that members of the NE were interested in developing a commercialized version.

During the interaction between the SEG and the NE, a sequence of six testing steps emerged. For the software engineer, the techniques used in each step are well known. However, there is no literature on the outcome of practical applications of these techniques to scientific software. The specific choice of technique or order of the steps is not as important as the idea of having a set of techniques as a "toolkit" where the requirements for use and the possible outcome of use of each technique is well understood. This case study contributes to this understanding. The testing strategy yielded far more than just a set of test data. Problems were revealed for the NE, in consistency of input data sets, effects of compilation options chosen, existence of silent failures, issues of reproducibility, sensitive numerical formulations, corrupt data files, and high coupling of supposedly independent modules.

Section 2 looks at existing literature on software engineering for scientific and engineering software developers. Section 3 describes the software environment for this study. Section 4 provides an outline of the testing strategy that was developed. Section 5 discusses each step of

the strategy, the required involvement of the software engineer and the scientific software developer, and the outcome of the application of the step. Section 6 concludes.

2. Interaction between Software Engineering and Scientists/Engineers Developing Software

The origins of computing derive strongly from scientific computing. But, as noted by Vessey [19], the computing discipline separated itself from engineering and science disciplines in the 1960s. Despite the identification of computing as a separate discipline, scientific communities have continued computing work on their own, largely independent of today's mainstream computer science. It is only recently that research in software engineering has acknowledged in any significant way the development of software by anyone other than full-time software professionals. In 2005, the first workshop in "End-User Software Engineering" was held [16]. Segal [17] has particularly defined the "professional end-user developer" as those software developers who are scientists and engineers. In a case study[18], Segal looks at organizational and process issues surrounding the development of a sample of scientific software. Segal's work illustrates how little software engineering research focuses on solutions for the specific needs of scientists and engineers.

An IFIP Working Group in October 2000 addressed the issue of architecture of scientific software [2]. This produced a set of papers, which, among other things, identified the typical characteristics of scientific software developers, their software, and their development environments. This is relevant as it sets out context in which improvement in support for professional end-user developers is needed. Other work looking at software practices of academic scientific researchers [21] identifies major gaps in their software knowledge and makes recommendations to fill those gaps.

In our work described in this paper, we describe an end-user developer in a scientific discipline (nuclear engineering) that demands considerable mathematical training, who is competent with computer languages, and who works in a rich technical domain, part of which is modeled in the software being developed.

There is considerable work that can be done in integrating knowledge from the computing discipline into useful and usable techniques and strategies for other professional disciplines where computing is an integral part of the deliverables of that discipline. Interestingly, we found no other papers that described a mix of testing techniques specifically useful for software developed by professional end-user developers. The study described in this paper appears to be the first contribution to describing the experience and understanding of applying testing techniques to this class of software.

3. Description of the Environment

The players in this study belong to two research groups, one in the Department of Chemistry and Chemical Engineering at the Royal Military College of Canada (RMC), and the other in the Department of Electrical and Computer Engineering at RMC. Within chemical engineering, there is a subgoup of nuclear engineers (NE), one of whom was the primary professional end-user developer participating in the interaction described in this paper. Within computer engineering, there is a subgroup of software engineers (SEG), from which we drew the software engineering participants, and of which the authors of this paper are members.

The professional end-user developer, who is both the code author and the domain (nuclear) specialist, had developed a prototype software system used to estimate radiation doses from cosmic sources. We will refer to the software system as 'Cosmic'. As is typical [2], [17], the code author was a specialist in the particular application domain, but not a software specialist. He and his colleagues were interested in the question of what should be done to raise Cosmic to a higher level of (commercial) quality, and were willing to bring in software engineering expertise to assist. Because of their focus on coding, their initial idea of involvement by members of the SEG was just to "clean up the code", perhaps rewriting it in another language.

As is typical of most engineering and scientific software systems, the main interest of the code author is in developing and fine-tuning the models that are eventually represented in the computer code. The software models draw on knowledge from different domains, including the physical phenomena of interest, the engineering, mathematical, and empirical models of the phenomena, knowledge of finite mathematical representations of the models, and knowledge of computer languages and numerical solution techniques. The domain specialist documents the models and the phenomena, and views the code as an extension of that documentation. He does not document the software for its own sake. The domain specialist makes changes to the software to improve the model, not to address such issues as software maintainability or usability. The domain specialist tests to show the model is working, not to show how or when the software may fail. When the purpose of the software development changes, from a focus on research on models and implementation of the models in code, to use of the software beyond the research lab, a software engineering viewpoint can help guide the transition. The choice of emphasis on software engineering issues to be addressed depends in part on the new purpose of the software. In the case of Cosmic, plans were for the software to be integrated into a larger system that would be developed to monitor aircrew cosmic radiation doses.

The version of Cosmic initially considered by the SEG had a simple user interface that accepted input data and displayed the results of calculations. The code was small (under 5000 LOC), written in Visual C++, and structured as a simple layered architecture. The calculations were done by a largely self-contained calculation engine. The calculation engine accessed a set of six files that provided informational lists (such as airport codes) and empirical data needed for the calculations. The content of the data files was significant both in that the calculation was intimately dependent on the data being correct and that the amount of data was significant (for example, one file had nearly 34 000 individual pieces of numeric data). The user interface developed for research purposes would be replaced in any commercial version of the software with an interface that included links to personnel data. Since the value of any further development on the system would be compromised if the calculation engine did not work correctly or was difficult to change, the calculation engine was the primary focus of our software engineering efforts.

Based on previous success inspecting code developed by professional end-user developers [12], [13], the SEG initially offered to carry out a code inspection of small samples of Cosmic. Six military officers (captain and majors) with several years of software experience took part in the initial inspection, recorded their findings, and met with members of the NE to discuss the inspection results and make recommendations. Because none of the military officers had a background in nuclear engineering, they were instructed to inspect for software engineering issues. They were supplied with hard copies of the source code for Cosmic, the User Manual

written by the NE, and several research papers describing the cosmic radiation model implemented in Cosmic.

The results of the inspection prompted the NE to ask for further involvement with the SEG. One of the authors (Côté, hereafter referred to as 'the software engineer'), who had been one of the inspectors, offered to do further work with the NE. During discussions with the NE, a number of areas had been identified that needed to be addressed, including configuration management, documentation, and installation configuration, as well as code restructuring. A decision was taken to concentrate on developing a test structure to support future work on the code, particularly eventual code restructuring. The software engineer saw a need to broaden the testing done by the professional end-user. A testing framework was developed with that purpose in mind.

4. A Prototype Testing Strategy for Professional End-User Developers

The term "testing" sometimes is limited to activities related to the dynamic operation of the software but is often expanded to include any activity that provides an evaluation, whether the evaluation is dynamic or static [9], [10]. Here, we use the second definition of testing, meaning evaluation in a broad sense.

When the software engineer began her work with the NE, the following resources were available: an existing software system, existing validation test cases, full access to the source code, limited documentation but access to people with appropriate knowledge of the system.

The strategy was developed on an iterative basis. Each testing technique in the strategy is well established in the software engineering literature but has not necessarily been evaluated for effective use with scientific software. We found that each technique in our strategy proved to be instrumental in providing different information for our assessment of the software, its data, and its environment.

Each step of the testing strategy required development effort from both the software engineer and the professional end-user and yielded feedback for both groups. Following is an outline of the steps in the strategy as it was developed in the Cosmic context. Section 5 provides details. Further discussion of the work can be found in [4].

1) Do a software inspection: The usual main goal of software inspections is to identify defects. Here, there were two important additional benefits. First, a broad inspection-based evaluation of Cosmic convinced the code author that concerted work with the software engineer would be beneficial. Second, the inspection provided a context for the software engineer to learn background information about the relevant radiation models.

2) Set up a regression suite containing only validated reference tests: This step methodically organized, catalogued, and executed the tests already run by the code author. As well as validating the validation tests, this activity provided a context for the software engineer to learn about the dynamic side of the radiation models

3) Build/Choose a test automation framework: In parallel with step 2, a framework with a user interface for testing was created to allow both the software engineer and the nuclear engineer to select which tests to run, determine the inputs to be used, and to flag test failures. This step also revealed deficiencies in the software structure related to efficient execution and module coupling, since the framework had to integrate closely with the software being tested.

4) Explore coverage criteria and augment the regression suite: Using coverage criteria as a systematic approach to augment the regression suite of tests, this step caused more thorough exploration of the ranges of input data and the software's response to invalid input. Graphs produced by the software engineer helped the code author visualize untested data ranges and redundant test cases in the set of tests available at the time this step was initiated. This led to a sequence of joint decisions on the best tests to be added.

5) Create a basis for choosing subsets of the full regression suite: To allow code changes to be tested quickly but retain a level of confidence in the tests, a systematic basis was developed for choosing small subsets of the reference test set. This step is only necessary when the whole regression suite takes a long time to run.

6) Test internal behaviour and set up automatic checking of results: Because calculation outputs are not exact, automatic checking of test results against expected results required careful discussion around the question of "What is close enough?" Tests were also added to check internal code behaviour for specific changes to increase confidence in the changes.

5. Details of the Development of the Testing Strategy

Developing the strategy and applying it to Cosmic provided valuable data on the quality of the software. The software engineer was able to provide detailed and important information for the code author related to the code, its associated data files, and existing test data. There was a significant learning curve for the software engineer as she gained a sufficient understanding of the radiation models implemented in the code, the context of their development and the testing that had already been done, as well as the future intended use. The entire exercise extended well over a year. The following sections describe what unfolded as each step of the strategy was implemented.

5.1 Do a Software Inspection

No part of Cosmic had been previously inspected. This is not unusual since inspections are not widely practiced in industry or research despite long-standing evidence that software inspection is the most effective quality assurance activity (eg. [20]). Given that the code author had no previous experience with inspection, he was not sure of its value as applied to Cosmic. An initial inspection of about 10% of the code revealed enough issues of interest to the code author that he requested that the inspection be expanded to include the entire calculation engine. When completed, the inspection yielded over 280 unique findings for about 2200 lines of inspected code. Since all the inspection was carried out by software engineers, the issues related primarily to maintainability, understandability, performance, documentation (or lack thereof), usability, and data integrity. Ideally, inspectors should be people with backgrounds that cover all the disciplines represented in the software [15]. A partially satisfactory alternative is to depend on the code author (domain expert) to double check any areas where the inspectors raised possible issues related to the discipline. It is an interesting open question as to whether this is as effective as inspection by a domain expert, particularly one guided by a set of questions as proposed in [15]

Besides identifying specific problems, the inspection provided data for discussion and opportunities for learning on both sides. The software engineer gained substantial understanding of Cosmic both during the inspection and in subsequent discussions with the code author. The

code author better understood the priority and scope of work required to raise the quality of the software to commercial expectations.

5.2 Set up a regression suite containing only validated reference tests

This step in the testing strategy creates a first version of a regression suite to check external behaviour of the system, using existing tests, which are gathered together, documented, organized, and validated. The main purpose of this step was to convert an informal (even casual) regression testing practice into one that is disciplined and repeatable.

The code author had tested Cosmic, comparing its results against hundreds of measurements taken in the physical world. This gave him confidence that the models programmed in Cosmic were correct and were correctly coded. However, the results of these tests had not been fully documented, and the tests themselves had generally been created to be run once, and not saved. To start creating a reference test set, all the code author's tests had to be recreated and saved, rerun, and the results captured. Extensive help was needed from the code author to select or recreate the test data and document the tests. The software engineer set up scripts and logs for the test runs.

An important activity in this step was the validation of the results of the reference tests. Typical of applications developed for knowledge rich domains [2], [17], [18], there are no detailed requirements specifications. Acceptability of test results is determined by the judgment of the domain specialist. Results are typically compared to calculations done by other means (eg. by hand or from another domain specific piece of software) or to real world measurements (eg. measurements from observed phenomena or from instrumented laboratory test rigs).

The code author was asked to inspect the results of the tests that were rerun as intended reference tests, comparing them to known results from past work with the software. This may seem like a redundant step, but we found that some of the intended reference test results were different from past results by as much as 10,000%. In the extensive investigation that ensued, we first verified that the source code we were using was indeed the right version. What came to light was that

- some reference tests were missing data,
- some variables were not being initialized correctly,
- there were inconsistencies in the use of optimization and debug flags in creating previous versions,
- some of the calculations were more sensitive than realized,
- data was used by the calculation engine that had been previously believed to be ignored,
- corrupted information existed in the look-up tables used by the calculation engine.

Until these problems were fixed, a validated reference test set could not be produced. This step had the greatest unexpected time to complete and uncovered more surprises for the code author. It also provided an excellent opportunity for the software engineer to further understand the dynamics of the system.

5.3 Build/Choose a Test Automation Framework

It is well known that initial design and coding of automated tests normally incurs a significant cost over and above that of running tests manually [5], [6]. This makes it important to plan and start test automation as early as is feasible to maximize return on effort.

We started this step in the strategy in parallel with setting up the regression suite. The test automation framework became an essential contributor to progress, once doubts were removed about whether it was a contributor to problems being experienced in the previous step.

Because the test automation framework is intended for use by the code author and subsequent professional end-user developers, its user interface must provide access to the actual unaltered user interface of the software system, as well as provide additional functionality for the needs of testing. This includes being able to select and run individual tests, groups of tests, or all tests. Other functionality includes adding and deleting tests, editing expected results, and determining the range of valid expected results. During development, professional end-user developers typically run their software using manually entered input values, examine the output for correctness, then run the software again using other input values. The main purpose of an automated testing framework applied to computation intensive software is to exercise the software much more thoroughly than is possible manually. The automation framework will typically compare results to the reference set and produce a report of differences. For some types of changes (eg. renaming identifiers, changing comment lines), the comparisons should yield exact matches. For other cases, depending on the sensitivity of the calculations in the software, significant differences can occur. The domain specialist needs to be consulted to discuss the changes that have caused differences and, as in step 2, determine the magnitude of differences that can be tolerated.

Because the software is executed more intensively during automated testing than it is during development, testing efficiency becomes important. There may be factors that can be readily changed to increase efficiency. In our case, each execution of the software opened and closed a set of files. A judgment was made that maintaining this particular pattern of manipulation of the files was not key to the application. The testing framework was therefore set up to open the files once at the beginning of a set of tests and close them at the end of the last test in the set.

In the process of developing the framework, the software engineer became aware that the user interface of Cosmic and the calculation engine were too strongly coupled. The test framework could access the calculation engine, which was the primary focus of testing, only through the Cosmic user interface, both in moving data into the calculation engine and in accessing the modules in the calculation engine. This coupling also was a factor in the next step of the testing strategy, and it emphasized the need for design changes to Cosmic to assist its move to a commercial product.

A useful addition to the automation framework was calculating execution times for each test. In some applications, it may also be useful to calculate execution times for parts of the application. The execution times help in choosing subsets of the regression suite either for quicker test turn-around or for monitoring specific parts of the application.

The automation framework for Cosmic was developed iteratively as more types of tests were added. Care was always taken to keep the framework accessible and flexible, that is, designed so that the professional end-user developer could readily add more tests. Using a commercial off-the-shelf testing framework was considered, but rejected for a variety of reasons, including the

need to incorporate the Cosmic user interface, the need to control the opening and closing of files, and the need for flexibility in deciding on when test results were acceptable.

5.4 Explore Coverage Criteria and Augment the Regression Suite

Unless a software system is trivially simple, exhaustive testing is impossible. The software developed by professional end-users in their knowledge-rich domains is typically far from trivial.

A profusion of testing techniques is described in the software engineering literature, all intended to help find "what subset of all possible test cases has the highest probability of detecting the most errors [14]". Many of the techniques include a type of coverage criteria to guide both the generation of tests and the decision to stop testing. The software engineering testing literature related to techniques and coverage criteria is very extensive and diverse (eg. [3], [8]), meaning that it can be difficult for professional end-user developers to find information of value to them.

For Cosmic, the software engineer chose three coverage approaches for the purpose of augmenting the reference test set. The first approach was to set a statement coverage target. The second and third approaches used equivalence partitioning and boundary value analysis respectively, to force broader and more disciplined exploration of the input space. Presentation of the input space partitions and boundary values was done using graphs to make the options clearer to the code author.

This step triggered considerable further investigation of Cosmic, its data and program structures, and its responses to a variety of input.

Although statement coverage is a weak form of coverage, it is relatively easy to apply and is well supported by tools. Given that the code author had not used any coverage measures, statement coverage was a natural starting point.

Statement coverage tools helped to identify parts of Cosmic not exercised by the reference test set. The initial level of statement coverage was 61%. Those parts of the code that were not covered were investigated. In addition to unreachable code and error handling code that always show up in statement coverage exercises, additional parts of Cosmic were identified that the code author agreed needed to be tested. These findings were of considerable interest to the code author as a demonstration of a technique that might be useful to him in future development.

Binder [1] suggests that 85% statement coverage is a reasonable target. Since Cosmic was a relatively small system, we decided that 90% coverage was feasible. In the end, 97% coverage was achieved by the augmented regression suite.

During iterations between steps one and two of the strategy, it became apparent that Cosmic accepted wider ranges of values for many of its input data items than the code author had explored. This suggested that a data coverage analysis would be beneficial. Two data coverage techniques were used. Equivalence partitioning provides guidance in choosing a reasonable subset of test values. Boundary value testing exercises the extremes of input data ranges. Studies have shown that “a surprising portion of faults [turn] out to be boundary value faults” [11].

We added a risk component to this. We observe that there is debate in the testing literature about the extent to which effort should be expended on creating test cases based on inputs that

are known to be high risk. In some cases, it may be better to modify the software to reduce the risks. There were problem areas known to the code author but not specifically included in his test cases since the tests did not relate to specific validation cases. The code author was consulted to identify ranges of input data values that had caused problems in the past. He was provided with a number of graphs of input ranges of his data. An examination of the graphs allowed the code author to provide a set of test data that potentially was high risk of being handled incorrectly.

Using these approaches, an analysis of the initial reference data set was performed. A list was created of equivalence classes, significant boundary values, and risk related values for all potential input values for Cosmic. Input data in test cases from the regression suite were compared to this list. Graphs of values for individual input data items and combinations of input data items, taken from the reference test data set, allowed the software engineer and the code author to identify duplications and omissions in the allowable ranges for these data items. Information on duplications was used in the next step of the testing strategy. Information on omissions was used by the code author to investigate the creation of further test cases. At the same time, the code author was delighted to see how complete some data coverage was. This increased his confidence in his models.

Boundary value and equivalence class analyses both include investigation of invalid input data. Some invalid input data (including missing data) showed up in the graphs, which triggered more inspections of specific parts of the code.

Test cases were created to submit classes of invalid data to Cosmic. Several problems were discovered. One problem was the assignment of multiple meanings to a single input variable. As a result, range checking was not consistently applied and invalid values for some meanings of the variable were not trapped. Another problem arose from the fact that input checking was performed by the user interface of Cosmic and not by the calculation engine. Given that the user interface would be replaced in any commercial product, the lack of checks on input data by the calculation engine meant the calculation engine lacked robustness. A third problem was inconsistencies at boundary values, for example including or excluding zero in a range of values. A fourth problem was inconsistencies in variable type. In one case, an input value was limited to integer by the user interface of Cosmic, but the calculation engine accepted double precision.

Sometimes Cosmic simply crashed when given an invalid input. At least this makes a problem clearly evident to a software user. A more insidious response is silent failure. A particularly bad case occurred when a search that failed on erroneous input replaced the erroneous input by the first value in a look-up table, then continued its calculations. Other cases occurred when default values were used improperly and variables were not cleared and reinitialized when the program changed calculation modes. Detection of silent failures usually requires extremely close observation of results by the professional end user developer. Often they are missed.

The specifics of problems encountered in our case would differ in other software systems. However, the list above is illustrative of the kinds of problems that may be uncovered in any software by this step in the strategy, and illustrates the need for close collaboration between the code author and the software engineer.

This step in the strategy clearly complemented the inspection step. Indeed, there were multiple iterations between doing small inspections and more testing. The involvement of the code author was key both to help the software engineer and to receive complete feedback on what had been

found. The particular form of exploration of the use of coverage criteria as described was effective for Cosmic, but it is likely that different forms would be needed for other software.

5.5 Create a basis for choosing subsets of the full regression suite

The software engineer, in making changes to make Cosmic more robust and more maintainable, decided to follow the refactoring principle, that is, change a little, test a little [7]. This principle fits well with the typical way of working for professional end-user developers [2], [17]. The goal of the software engineer was to always work with a set of tests that executed within a given time limit. She combined this with her previous decision to attain at least 90% code statement coverage with any set of tests.

To demonstrate this step in the test strategy, the software engineer and code author selected an issue identified during the inspection, to be implemented as changes in Cosmic. For this set of code changes, the software engineer selected tests based on data coverage, using the equivalence classes set up in step 4 of the testing strategy. Execution times were included in the test scripts and were available to compare times for tests that had been flagged as duplicates in the equivalence class analysis. The shorter duration tests were always chosen first when creating a subset of tests. Failure of the set of tests to meet either the 90% coverage goal or the execution time limit caused iteration on the selection of tests. The regression suite and the software system itself were small enough that the iterations on test selection converged quickly.

At night, the software engineer ran the entire reference test set. One problem with badly formed equations was found by the overnight runs. Even though data analyses such as equivalence classes identify sets of numbers that are logically equivalent, it is not necessarily true that all numbers in the class are treated equivalently by the finite arithmetic. Badly formed equations, that is, equations that do not take into account the difference between finite and continuous mathematics, can cause large differences in expected outcomes. There were clear examples where equivalence class analysis of input data did not work reliably for Cosmic. This would be true of any computationally intensive software. Any cases identified in this way were added to the list of risky data values.

Again, the specifics of this step as followed for Cosmic would be different for other software.

5.6 Test internal behaviour and set up automatic checking of results.

Test cases created so far in the test strategy checked the external behaviour of the software. Tests created in this final step of the strategy check internal behaviour of the changed parts of the software. Before making changes, the software engineer must understand the behaviour of those individual (internal) parts, again possibly with the help of the code author. Tests are written to exercise the changed parts of the software and are added to the regression suite. Two approaches can be used to check the results of these tests. Results from the tests can be displayed in the automation framework for the software engineer and the code author to check manually. A second approach, which was used here, involves calculating and embedding expected results in the automation framework. The framework compares the test results to the expected results and reports pass or failure.

Behaviour of the code after changes may not be identical to that observed prior to the changes. This raises the question of how close is close enough. Answering the question required

cooperation between the software engineer and the code author. The software engineer can only point out inconsistencies in results from different versions of the software; only a domain expert can validate the correctness of the results.

Validity of the results involves two types of comparisons. One is between the computerized model and the physical phenomena being modeled. A domain expert must decide what is acceptable. The second comparison is between past and present results from the same software system. Ideally, the results would be identical. In reality, they seldom are. Rounding error, due to the finite nature of the computer, is a typical culprit and depending on the sensitivity of the numerical calculations, can cause large differences. This can lead to investigations of sub-optimal numerical techniques. In any case, tolerances must be specified for both types of comparisons, and the judgment of a domain expert is needed. Acceptable tolerances were embedded in the automation framework.

6. Conclusion

Two tangible products were established by our testing strategy: one was a flexible automated testing framework that could be augmented with new test cases by the professional end-user developer as software development and changes continued; the second was the reference test suite that was systematically validated by the professional end-user developers. An important byproduct of applying the testing strategy was the collection of observations made by the thorough examination of the software, the data, and the engineering models embedded in the software.

Perhaps more importantly, the application of the strategy showed how well know-how from the two disciplines, that of the software engineer and that of the professional end-user developer could be meshed to achieve a better software product. The professional end-user developer's initial concept of help from the software engineer was limited to code changes. The help provided by the software engineer proved to be far more extensive, systematic, and fundamental than expected. The software engineer however could not do this alone. The close cooperation of the two disciplines was key.

This work is a contribution to the literature on effective interaction by software engineers with software developed by professional end-users. There is much opportunity to extend this work to other collaborations, both to help other professional end-user developers, and to elaborate further on the nature of useful interactions.

7. References

[1] Robert V. Binder, Testing Object Oriented Systems: Models, Patterns, and Tools, Addison-Wesley Longman Inc., 2000
[2] Ronald F. Boisvert, Ping Tak Peter Tang, editors, The Architecture of Scientific Software, Kluwer Academic Publishers, 2001
[3] Antonia Bertolino, "Software Testing", Chapter 5, SWEBOK stoneman version, 2000 http://www.swebok.org/stoneman
[4] Nancy Cote, "An Exploration of a Testing Strategy to Support Refactoring", Master's Thesis, Royal Military College of Canada, April 2005
[5] Elfriede Dustin, Jeff Rashka, John Paul, Automated Software Testing, Addison-Wesley, 1999
[6]Mark Fewster, Dorothy Graham, Software Test Automation, Addison-Wesley, 1999

[7] M.Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2004
[8] Mary Jean Harold, "Testing: A Roadmap", ICSE 2000, The Future of SE Track
[9] IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology
[10] IEEE Standard 829-1998, IEEE Standard for Software Test Documentation
[11] Paul C. Jorgensen, Software Testing: A Craftsman's Approach, CRC Press, New York, 1995
[12] Diane Kelly and Terry Shepard, "Task-Directed Software Inspection Technique: An Experiment and Case Study", Proceedings IBM CASCON 2000, Toronto, November 2000
[13] Diane Kelly, Terry Shepard, "Eight Maxims for Software Code Inspections", Journal of Software Testing, Verification, and Reliability (JSTVR), Volume 14, Issue 4, December 2004, pp. 243 - 256, published online: 15 Jun 2004
[14] G.J. Myers, T. Badgett, T.M. Thomas, C. Sadler, The Art of Software Testing, 2nd Ed. John Wiley& Sons Inc., 2004
[15] David L. Parnas, David M. Weiss, "Active Design Reviews: Principles and Practice", Proceedings 8th International Conference on Software Engineering, London UK, August 1985, pp. 132-136
[16] Gregg Rothermel, Sebastian Elbaum, "The First Workshop on End-User Software Engineering", Proceedings ICSE'05, May 2005, p. 698
[17] Judith Segal, "Professional end user developers and software development knowledge", Technical Report no. 2004/25, Open University UK, October 12, 2004
[18] Judith Segal, "When Software Engineers Met Research Scientists: A Case Study", Empirical Software Engineering, 10, 2005, pp. 517-536
[19] Iris Vessey, "Problems Versus Solutions: The Role of the Application Domain in Software", Papers presented at the 7th Workshop on Empirical Studies of Programmers, 1997, ACM Press, New York
[20] David A.Wheeler, Bill Brykczynski, Reginald N. Meeson, Jr.; Software Inspection An Industry Best Practice, IEEE Computer Society Press, 1996
[21] Greg Wilson, "Where's the Real Bottleneck in Scientific Computing?" American Scientist (01/06) Vol. 94, No. 1, p. 5