Academic vs. Industrial Software Engineering: Closing the Gap

Andrey N.Terekhov[1] and Len Erlikh[2]

[1] St.Petersburg State University, LANIT-TERCOM
Bibliotechnaya sq., 2, office 3386
198504, St.Petersburg, Russia
ant@tercom.ru
[2] Relativity Technologies, Inc.
1001 Winstead Drive
27513, Cary, NC, USA
Len.Erlikh@relativity.com

Abstract. We argue that there is a gap between software engineering cultivated in the universities and industrial software development. We believe that it is possible to get academia and industry closer by starting projects that will require solution of non-trivial scientific tasks from one side and long-term commitment to create a product out of this research solutions from the other side. We illustrate our position on a real-world example of collaboration between an American company Relativity Technologies and research teams from St.Petersburg and Novosibirk State Universities. We also point out that the current economic situation in Russia presents unique opportunity for international projects.

Introduction

Industrial programming is usually associated with big teams of programmers, strict timelines and established solutions and technologies. On the other hand, the main goal of academic research is to find new solutions and break existing stereotypes. Unfortunately, amazingly low percentage of scientific results makes their way into practice, and even when they do, the process is very slow.

In the meantime, practice always required a solution of the tasks that are infeasible from the point of view of the existing theory. Today this common truth takes on special significance for software engineering because the number of its applications really exploded. A special emphasis on this problem was made by academician A.P. Ershov. By the way, it is little known that for several years A.P. Ershov worked as a consultant in research institute "Zvezda" of LNPO "Krasnaya Zarya" (Leningrad). By tradition of those times this job was not paraded, because the institute worked in the area of government communications, so later on even specialists who knew A.P. Ershov personally, were surprised that a scientist so famous was spending his time on such "utilitarian" problems.

It is a well-known situation when practitioner is posing a problem and theoretician is reasoning why this task is unlikely to be solved. But the proof of

impossibility of correct solution of the problem does not satisfy the demand for it, so practitioners start to seek partial or heuristic solutions or try to use "brute force" method.

In this article we try to show that even on this shaky ground it is better to use specialists that know the theoretical restrictions, complexity estimations for this or that solutions, optimization methods and other traditionally scientific knowledge. This sounds pretty obvious, but somehow the chasm between academic and scientific communities is very difficult to close. What are the main reasons for this?

It is well-known that software engineering is differing from pure mathematics or even computer science. Proved theorem or complexity estimation for some algorithm are results by themselves, and there are no other requirements for their creation other than scientists' talent, pen and paper. On the other hand, in software engineering a new interesting approach or even working prototype does not guarantee that they will lead to the successful and ready-to-use product. To achieve this, one should add up large teams, investments and strict industrial discipline. In this respect, software engineering is close to elementary-particles physics or physics of ultralow temperature etc.

Nevertheless, there are some positive examples and we believe that they could be considered as role-models for promoting collaboration between industry and science. We try to illustrate this process on the example of creating an automated reengineering tool RescueWare, which automates reengineering of legacy software, i.e., conversion of systems written in COBOL, CICS, embedded SQL, BMS, PL/I, ADABAS Natural and other languages, working mostly on IBM mainframes to C++, VB or Java. Software reengineering does not end up in simple translation from one language to another — completlely different schemes of dialog with the user, access to legacy databases, recovery of lost knowledge about the program make this task much more difficult.

This project was carried out by large international team, which was geographically spread from North Carolina (USA) to St.Petersburg and Novosibirsk. The customer for this project was an American company Relativity Technologies, and the team that worked on this project included scientists from S.-Petersburg and Novosibirsk universities, LANIT-TERCOM company and Institue of Informatics Systems of Siberian division of Russian Academy of Sciences. The total investment in this system amounts to more than 400 man-years.

Our collaboration began in 1991, and during these years we have overcome a lot of difficulties, mostly related to cultural difference and lack of understanding. Finally, as a result of common efforts we created a science intensive product RescueWare Workbench, which was recognized by Gartner Group as a best product in 2000 in the area of legacy understanding. In the following sections we give a detailed descriptions of the hard problems that we had to solve in order to make this project a success.

1 Architecture for multi-language support

From the very beginning we were oriented on creation of *multi-language* translator, so one of our first tasks was to design a unified intermediate language (IL) for our system. The idea is that program transformation is two-staged: at first the program is converted to IL, and then to the target language. When there are M input and N output languages, this approach makes it possible to limit the amount of work to $M + N$ compilers instead of $M * N$.

Under the name of UNCOL this approach is known for more than 30 years. A lot of teams were trying to implement it, but only a few managed to create something tangible [1, 2]. Now what is the problem? From the theoretical point of view, all langugages are computationally equivalent, and thus conversion should be quite simple. The difficulties arise when we are measuring the quality of output program not only from the point of view of its performance, but mostly from the point of view of naturality of program's structure in this or that language.

Every programming language gives us some means of expression and a discipline of using them. At that, the most quality from the point of view of quality and easiness of maintenace could be attributed to those programs that meet the requirements of this discipline.

The problem appears when notion of "natural program" in one language contradicts with the same notion in other language. For instance, using GOTO statements is usual for COBOL, but in Java there are no such statements at all. Some special and sometimes non-trivial transformations, dependent both on initial and target platforms, may be required to solve this problem.

Let us name the main levels of program representation:

1. Control flow representation
2. Data flow representation
3. Representation of values

Thus IL must contain abstract means for program representation at all these levels, and the transformation will look as follows: first language constructs of the source language are "raised" to abstract intermediate representation and then they are "lowered" to concrete representation in the target language. The degree of abstraction should be carefully chosen to make sure that this lowering down leads to natural projections to the target language.

The most important condition for the proposed approach is the *orthogonality* of translations to IL and from it. This means that the representation of source program in IL should not depend on the target language.

Finally, IL should be extendable, i.e., it should contain features that permit to build new constructs without changes to all existing passes of compiler.

The weak point during IL design is the choice of data types and standard opertions. While the set of control constructs in different languages is more or less suitable for unification, data types system could be significantly different. It is inexpedient to simply combine all types of the source languages, because addition of new language will require major changes to existing compiler functionality.

To solve this problem, it is important to add not only standard data types to IL, but also add formalism of higher order — *type constructor* — which could be regarded as a function, which produces a new type out of several given types. For example, abstractions such as structures, arrays and pointers could be considered as type constructors. Indeed, the structure could be defined as a type constructor, which receives a set of field types and creates a structured types with the fields of corresponding types.

Finally, usage of type constructors eases runtime support, for we can consider type constructor, which is treated as an abstract dynamic data type and could be used only through runtime support functions. This ensures that the addition of a new entity requires changes in only one compiler pass — namely, of the pass, which creates this entity. After that all handling of this entity will be transparent to the compiler and conducted through conversion of type constructors.

We believe that this task presents a good example of semantic gap between academic research and industrial programming. The idea of unified IL makes sense only in large-scale projects, and these projects are out of acaedimic scope. On the other hand, average programmer in the industry just does not possess all the knowledge, which is required for successful implementation of this approach.

Note that "naturality" of IL structure, which was mentioned above, is also a good example of difficult to formalize notions. These notions are often necessary to solve usual everyday tasks. Another example of difficult to formalize notions is the definition of "good program" criteria [3].

2 Re-modularization. Class Builder

Another interesting task that we encountered during creation of automated reengineering tool is re-modularization of programs into components [4, 5].

This task could be described as follows: there is a large application that consists of multiple files, which contain declaration of data and procedures. Variables and procedures from different files are interacting with each other through some external objects, which we called *dataports*. For legacy systems usual dataports are CICS statements, embedded SQL and other infrastructure elements.

This task was formalized as follows: application was represented as a graph with application objects as junctions of various types (variable, procedure or external object), and relations between them as graph edges. Edges are also typed (for instance, procedure call, variable usage in procedure, working with external object through variable etc.). Also, each edge is attributed with some number, which defines the "power" of this relation. For example, the power for procedure call relation could be defined by the number of parameters passed: the more parameters we have, the more we want to place both callee and caller to one component.

This graph should be divided into some areas of strong connectivity. To do this, we introduce the notion of gravity between two nodes, which is calculated as sum of powers of all edges connecting them multiplied to coefficient defined

by the edge type, minus some constant, which is defined by the pair of edge types.

Some negative part of the formula — the repulsive force — should be included, otherwise we will always get one monolith. For instance, it seems natural to add repulsive forces between dataports and any procedures. This way we want to separate all procedures first and only when two procedures are using too many common variables, then the gravity force prevails.

Then by complete enumeration we find those junction sets, for which the sum of gravity force between themselves and the junctions from other sets are maximal (of course, gravity forces with the junctions of other groups are taken with minus sign).

It is clear that this good idea will not work in real life, because the number of graph junctions in real applications is way too much to use exhaustive searches. But we managed to find some heuristic approaches, which made it possible to achieve practical results.

First of all, we fixed some coefficients for different types of edges and repulsive forces for different types of junctions. However, the user can assign coefficients on his own if he believes this to be of importance for his application.

Secondly, in the complete graph of application we will start with sub-graph, which consists of the junctions corresponding to external object plus edges and junctions of any other types, which connect these external objects. The heuristics is that we believe external objects to be cross-linking and thus we add repulsive forces only for them. On the other hand, if two external objects are using a lot of common variables and procedures, then nothing prevents them from ending up in one component.

Thirdly, we will regard all edges of reduced graph as being of the same type, but we will define the power of each edge so that the more relations there are (not only direct, but also transitive ones), the bigger is power. To formalize this notion of "bigger", we will use the physical model.

To calculate the power of edges in reduced graph we proposed to use the model of electric mains. In this model we will consider that there exists a wire between any two junctions of the input graph that have a positive gravity force between them and we will equal this force to the conductivity of the wire (let us remind that conductivity is the reverse function of resistance). Then as a power of edge between two junctions of the reduced graph we can use the complete conductivity of the electrical network between them. The complete conductivity could be calculated using Kirhgof equations, so we need to solve a set of linear equations. The complexity of this task is equal to finding the inverse matrix.

3 Program Slicing

Let us suppose that a legacy system performs ten functions, seven of which are no longer needed, but the remaining three are in active use, and as it often happens with legacy programs, nobody knows *how* these three functions work. In this case it is necessary to create a tool for deep analysis of the old programs, which

can help maintenance engineer to find and pick out the required functionality, put the corresponding parts of the program into a separate module and reuse it in the future, for instance, to move it to modern language platform.

The solution of this task is based on creating static slices of the program and their modifications. We regard program slices to be a subset of program statements that presents the same execution context as the whole program. Slice is a program that contains the given statement and some other statements of the initial program, namely those that are related to this statement.

The following methods are implemented in RescueWare for automation of business rule extraction (BRE):

- Computational-based BRE
- Domain-oriented BRE
- Structure-oriented BRE
- Global BRE

All these methods assume generation of syntactically correct independent programs that preserve the semantics of the original code fragments.

Computational-based BRE forms the functional slice of the program, based upon the execution path and data definitions that are required to calculate the values of the given variable in the certain point of the program [9]).

Domain-oriented BRE generates functional slice of the program, which is received by fixing the value of one of the input variables. Being based on theory of program specialization, domain-oriented BRE is best suited to separate calculations with many transactions and mixed input, into a series of "narrowly specialized" business rules with only one transaction for each of them.

Structure-oriented BRE makes it possible to divide the programs written as a single monolith into several independent business rules, based on the physical structure of the initial program. Also, an additional program is generated that calls the extracted slices in a proper sequence and using the correct parameters (ensuring the semantic equivalence of this program to the initial one). This method is best suited to divide old large programs into parts that are easier to handle.

Finally, global BRE helps to apply all of the methods mentioned above to a number of programs simultaneously, and thus supports BRE on system-wide basis.

Notwithstanding all automation, the choice of slicing points in the program and the sequence of application of different BRE methods are left to the human analytic, which decomposes the initial system. There are several natural points to start applying BRE, for instance, the places where the calculated values are stored to the database or printed to screen. Of course, intelligent choice of candidates for business rules requires knowledge about functions of the initial system.

On closer examination it often turns out that the methods used are differing from one module to another. Moreover, it is sometimes useful to apply different BRE methods subsequently to the same program.

4 Conclusion

As of right now, products such as RescueWare are not really typical for the market, because creation of RescueWare required solution of *many* scientifically difficult problems. Let us briefly mention other achievements: relaxed parser that ensures collection of useful information even for quite distant dialects of the language, different variants of data flow analysis, using sophisticated algorithms of pattern matching for identification of structure fields in PL/I etc.

Of course, not all projects require investment of this amount of research. Even in our own practice not all projects both in Russia and USA were so rewarding. Our present understanding of importance of education and training in software engineering came only after a lot of painful experience. The contacts with the universities are important for the industry not only because of the talent pool accumulated there, but also because of new ideas and scientific breakthroughs that can make the sofware product a success. This road is very difficult; sometimes the scientific research contradicts with the strict timelines and discipline, but the potential payoff of this approach is immense.

Finally, we would like to emphasize that Russia is in a special position to make this vision come true, because it has an undoubted advantage in the level of education at the software market. We hope that our experience of successful cooperation of American company with Russian scientists could serve as a good example for many Western companies.

References

1. A.P. Ershov “Design specifications for multi-language programming system”, Cybernetics, 1975, No. 4. P. 11–27 (in Russian)
2. GCC home page. http://www.gnu.org/software/gcc/gcc.html
3. I.V. Pottosin “A “Good Program”: An Attempt at an Exact Definition of the Term” // Programming and Computer Software, Vol. 23, No. 2, 1997. P. 59–69.
4. A.A. Terekhov “Automated extraction of classes from legacy systems” // In A.N. Terekhov, A.A. Terekhov (eds.) “Automated Software Reengineering”, S.-Petersburg, 2000, P. 229–250 (in Russian)
5. S. Jarzabek, P. Knauber “Synergy between Component-based and Generative Approaches”, In Proceedings of ESEC/FSE’99, Lecture Notes in Computer Science No. 1687, Springer Verlag, P. 429–445.
6. M. Weiser “Program Slicing” // IEEE Transactions on Software Engineering. July 1984. N 4. P. 352–357.
7. T. Ball, S. Horwitz “Slicing Programs with Arbitrary Control Flow” // Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging. 1993.
8. M.A. Bulyonkov, D.E. Baburin “HyperCode — open system for program visualization” // In A.N. Terekhov, A.A. Terekhov (eds.) “Automated Software Reengineering”, S.-Petersburg, 2000. P. 165–183 (in Russian)
9. A.V. Drunin “Automated creation of program components on the basis of legacy programs” // In A.N. Terekhov, A.A. Terekhov (eds.) “Automated Software Reengineering”, S.-Petersburg, 2000. P. 184–205 (in Russian)