8/1 (2010), 133–148
tmcs@math.klte.hu
http://tmcs.math.klte.hu
Teaching Mathematics and Computer Science

Software engineering education in cooperation with industrial partners

Zoltán Horváth, Tamás Kozsik and László Lövei

Abstract. This paper presents our experiences on teaching software engineering in teams which are organized around different R+D projects. These long-running, innovative projects are carried out in cooperation with industrial partners, and are supported by student exchange. While MSc and PhD students work together with faculty staff members on the projects in an industrial-like environment, the students develop skills that would be otherwise very hard for them to obtain. The methodological contributions of the paper are illustrated by, and substantiated with, the description of a concrete software engineering project.

Key words and phrases: software engineering education, team work, research and development projects.

ZDM Subject Classification: P55, P65, Q55.

1. Introduction

This paper presents our experiences on a software engineering course organized around different R+D projects for students working in research teams. Involving students in research projects and requiring students to work in teams are widely used methods in software engineering education. However, such projects often remain at the level of a university excercise both in size and in complexity. The projects described in this paper are not of that kind. They have been running for years and yield large software systems to maintain and extend. They are

Supported by ELTE IKKK and Ericsson Hungary.

carried out in cooperation with industrial partners, and are supported by student exchange. Furthermore, MSc and PhD students and faculty staff members work together on the projects in an international and industrial-like environment. Under these circumstances the students develop skills that would be otherwise very hard for them to obtain.

A major advantage in using real industrial projects in education compared to general university exercises is the growth of collective and individual responsibility in maintaining high quality and respecting deadlines. The fact that the results of these innovative projects are used in the software industry is highly motivating for the students.

The first project started 3 years ago. The project, aiming at the development of a refactoring tool for the Erlang programming language, is supported by Ericsson Hungary, and it is being carried out in Erasmus cooperation with University of Kent, University of Sheffield and Erlang Training and Consulting.

The paper is structured as follows. Section 2 gives an overview of the projects involved in the course. In Sections 3 to 7 one of the projects, namely the one about refactoring Erlang programs is used for illustration purposes. Section 3 summarizes the relevant curriculum. It sketches the preceding courses, explains how the project course fits in the curriculum, and illustrates how the project results can be used again to improve education. Section 4 describes the methodology applied during the course, and how the project team is organized. Section 5 discusses how to design the structure of the software developed during the project in order to facilitate the definition of tasks for the students. Section 6 enumerates the skills students develop while working on the project. In Section 7 the results obtained during the project are presented. Finally, Section 8 addresses related work and concludes the paper.

2. The projects in the course

Currently five projects are available for our students to take part with in the course, three of them are about industrial application of functional programming. In each project about 10–15 MSc and PhD students work together with 2–3 faculty staff members. The topics of the projects need to be selected very carefully: they should be suitable for education and also for research, so that all the different team members benefit from the projects. The different topics are the following.

Refactoring Erlang programs: As mentioned earlier, this project aims at developing a refactoring tool for Erlang. The tool supports static program analysis and semantics preserving program transformations [15].

Analysis of F# Programs: This project is also related to static program analysis. The goal is to develop tools enhancing programmer's efficiency in F# [7].

A domain specific language for DSP: In this project a high-level domain specific language designed for digital signal processing algorithms is being developed, and a prototype compiler and related tools (e.g. debugger) for the language is being implemented [8].

HypereiDoc: This project is targeted at the development of an XML based framework supporting distributed, multi-layered, version-controlled processing of epigraphical, papyrological or similar texts in a modern critical edition [9], [2], [20], [33].

Mobile ODF: Viewing and editing documents in OpenDocument Format requires the amount of resources (memory, CPU and power) that is usually not available on mobile platforms, e.g. on mobile phones. This project develops the technology for the management of ODF documents on phones running Java ME: the definition of schemas that can be adapted to the capabilities of given mobile platforms, the design of schema and document conversions and the implementation of a supporting client-server software infrastructure.

Most of the above projects are carried out in collaboration with partner universities and companies utilizing Ceepus student exchange, Erasmus university exchange and Erasmus industrial training student placement. Apart from the international partners, the projects are supported also by local companies, such as Ericsson Hungary and Morgan Stanley Hungary.

3. The curriculum

In the following sections one of the projects is described in details. This project is aimed at developing a refactoring tool [10] for the Erlang programming language [1]. Erlang is an eager, impure, dynamically typed functional programming language developed by Ericsson. It was designed for building concurrent and distributed fault-tolerant systems with soft real-time characteristics, like telecommunication systems. The Erlang language consists of simple functional constructs extended with message passing to manage concurrency. Erlang has a module system with export/import lists, exception handling, reflective programming facilities, preprocessing mechanism to support macros and file inclusion, and a comprehensive standard library.

Before learning Erlang, our students complete a number of courses related to programming languages and software engineering. Some of these courses provide foundations: formal methods for the specification and synthesis of correct sequential, parallel and distributed programs, compiler construction, algorithms and a strong mathematical background. The students learn imperative (C++, Ada, Java) and functional (Clean or Haskell) programming languages for three semesters. In these courses they obtain comprehensible knowledge in the concepts and constructs of programming languages including concurrency (Ada tasking) and message passing (PVM), but no experience with the development of large software systems. Other courses provide standard material on software engineering [34] – the practical part of these requires project work of small teams (with 3–4 members). These projects are at usual university exercise level, not reaching the amount of complexity typical for an industrial project. At master level motivated students are encouraged to take optional courses on type systems and compilation of functional languages, lambda calculus, distributed and parallel functional programming, formal semantics and different proof tools.

3.1. The Erlang course

One of these optional courses, gaining increasing popularity, is about programming in Erlang. We started this course after the proposal of Ericsson Hungary (they develop telecommunication applications in Erlang) and a successful summer school course (held by Ericsson and Erlang Training and Consulting) at 1st Central European Functional Programming School [14]. The course is available for master students as well as for last-year bachelor students. Currently the majority of the students taking this course have been registered for our now obsolete one-track 5-year master programme with an optional BSc degree obtainable any time between the 3rd and 5th years. In the first year, 3 years ago, we had 3 participating students. Since then we have in average 20 students a year, about the half actually completing the course.

The Erlang course focuses on the dynamic nature of Erlang, on processes and message passing and on the "behaviour" patterns of the Erlang/Otp library. The Erlang/Otp patterns (genserver, finite state machine, supervisor) help managing software complexity.

3.2. The RefactorErl project course

The students that turn out to be the best at the Erlang course are invited to take part in a research and development project aiming at the design and implementation of RefactorErl, a refactoring tool for Erlang [15]. This project work serves as an advanced course on software engineering and programming in the large. The students are offered 16 ETCS credits for this course. Furthermore, most of the students continue the project as a BSc and/or MSc thesis work.

The overall size of the software that the students work on is around 30 kLOC in Erlang. Our impression from comparing this project to the Hyperei-Doc project [9] is that it is very hard to manage a similar project with students using languages like Java: functional programming principles really help cope with the complexity of RefactorErl.

3.3. The RefactorErl tool in the education

Refactoring is the process of applying semantics preserving transformations ("refactorings") on a program in order to improve its quality or to prepare it for forthcoming changes [10]. The use of a refactoring tool can facilitate this process by performing tedious or error-prone activities automatically and safely. In fact the process may involve many transformations applied by the refactoring tool, interleaved with the manual edition of the program text by the programmer(s). The refactoring tool is usually integrated into the program development environment, and this makes it easy for programmers to apply a refactoring transformation whenever required during the coding phase.

Interestingly, a refactoring tool is useful not only in software engineering, but also in education. This concept was proved on 2nd Central European Functional Programming School [17], where the first prototype of RefactorErl was used to teach the depth of Erlang syntax and semantics as well as proper coding techniques and styles for 35 students [21]. At the beginner level, a refactoring tool can help the students understand how to improve the quality of the code in a step-by-step manner towards a concise, elegant, functional style, and also how to make the code suitable for the addition of new features.

4. Methodology

The R+D work on RefactorErl is an industrial-like project supported by Ericsson Hungary. A workplan is set up twice a year, and further communication with the company takes place on the requirements on the tool on a weekly basis. The second version of the tool has been applied by Ericsson successfully for reorganizing the module structure of a large body of code of a complex telecommunication software [25], [22].

Our project team is built up of 3 faculty staff members, 2–4 PhD students (with related PhD research topic and SE experience) and 7–12 master students, varying over time. Changing the size and the members of the team, is due to two reasons. Firstly, students often spend a semester abroad as exchange students, and secondly, new students join the team in every semester. This fluctuation can be tolerated only if the internal structure of the developed software is very carefully designed, and the interfaces are very precisely defined and documented. The lessons learned from the development of the first prototype of the tool [30] showed us that the structure successful for 2–3 students became inappropriate for involving more students. Therefore we had to redesign and reimplement the tool from scratch. The resulting structure [16] makes it possible to define 1 person month tasks. Even in this well-defined structure continuous team-wise and peer-to-peer communication is required. Project meetings are organized twice a week (about 2 hours each).

The topics of the meetings are the review of the state of the project (analysis of the results obtained, tasks for the upcoming week), general discussion on long-term design (so that the new team members can catch up with the concepts, the overall structure, the interfaces and the general infrastructure of the system), invention of special algorithms for more complicated tasks (which often involves the evaluation of related work found in literature), and also regular code inspection.

Code inspection turned out to be very important: students (especially the beginners) often make bad design decisions, and produce code of bad quality, of bad (non-functional) style or of unacceptable efficiency. They tend to forget about coding conventions (improperly chosen identifiers, poorly written comments). In this process the responsibility of the project leader is high: with proper social and psychological abilities the leader can help avoid personal conflicts, and the team as a whole can profit a lot from many aspects.

- Members of the team develop *cooperative attitude.*
- The *critical analysis of each others' product* not only improves quality, but also enhances students' skills.
- The discussions result in a *coherent approach and the consistent structure* of the application.

- The *thorough knowledge of the inner structure* ensures that the code the students write fits well within the system, even though the students are rather unexperienced.
- Team members can *avoid code duplication.*

Twice a year a progress report is assembled. This also fosters knowledge transfer between generations of students.

5. Defining tasks for students

The development of the first version of RefactorErl showed us that team work involving students is possible only after designing a clear modular structure for the software. The structure of the second version, presented below, has a crucial impact on the ability of providing reasonable tasks for students.

RefactorErl represents an Erlang program as a "program graph": a directed, rooted graph with typed nodes and edges. The skeleton of this graph is the abstract syntax tree of the program. Apart from syntactical information, the graph contains lexical and semantical information as well. These latter are provided as additional (lexical and semantical) nodes and edges in the graph. For example, each function in the program is represented as a semantic node in the graph; the definition of the function and all the calls to the function are linked to this semantic node with semantic edges. The maintenance of semantic information is useful for boosting the checking of side conditions. Usually, the hardest part of refactoring is not the application of the requested transformation, but the evaluation of the conditions that are required to hold for the refactoring to be safe. These conditions often depend on a large amount of semantical information – which can be efficiently picked out from the program graph. Apparently, the stored semantical information, similarly to the AST, must be updated when a transformation is applied.

In order to improve the efficiency of refactoring large programs, the construction of graphs representing programs must be incremental: the graphs are persisted in a database, so a (sub)graph representing a module needs to be recomputed only when the module is altered manually (i.e. not with the refactoring tool). This approach is especially useful when a large number of refactoring transformations are to be applied on a program without intervening edition of the code by programmers. This happens, for instance, before introducing a new feature into an (otherwise fully functional) program, which may require substantial reorganization of the code.

Lexical information, such as the tokens produced by the scanner, is also essential in a production-level refactoring tool. Even information about the whitespace separating the tokens must be kept available so that the tool can preserve the layout of the refactored program.

The kinds of semantic information to be gathered and maintained by the refactoring tool depend on the transformations the tool itself supports. The RefactorErl tool is designed to be open-ended: it should be possible to implement further transformations with relevant semantical analyses and add them to the refactoring framework. To achieve this goal, the different kinds of semantical analysis are organized into independent modules, and these modules provide independent sets of semantic nodes and edges for the program graph. Examples of semantic analysis modules are the analysis of scopes, the analysis of function definitions and calls, or the analysis of variable bindings.

The tool also includes a query language, similar to XPath, for retrieving information from the program graph. Links of the graph can be traversed forwards and backwards, and filtering by semantical information is also supported.

To optimize the shape of the program graph for fast information retrieval, the syntax of the language is reflected in the tool at two levels of abstraction. In the more abstract view there are four syntactical categories: files, forms, clauses and expressions. Files (including header files) contain forms. Forms can be, among others, function definitions, which are made up of one or more clauses (clauses are basic building blocks of several compound expressions as well, such as `case`-expressions). The right-hand side of a clause is a sequence of expressions and the left-hand side of a clause contains further expressions such as patterns and guards. The rich syntactical structure of Erlang, reflected in the close to fifty rules of the grammar, can be abstracted into these four kinds of graph nodes. Many details of the syntax are encoded in the types of the graph edges, forming the less abstract syntactic view of the language.

The refactoring tool has a layered implementation. The overview of these layers is as follows (see Figure 5).

1. The lowest level layer provides persistent storage of labelled, ordered graphs that are valid for a given schema. Path expressions are implemented in this layer. The current implementation uses Mnesia, the standard DBMS for Erlang/OTP, to persistently store the semantical graph. An arbitrary graph

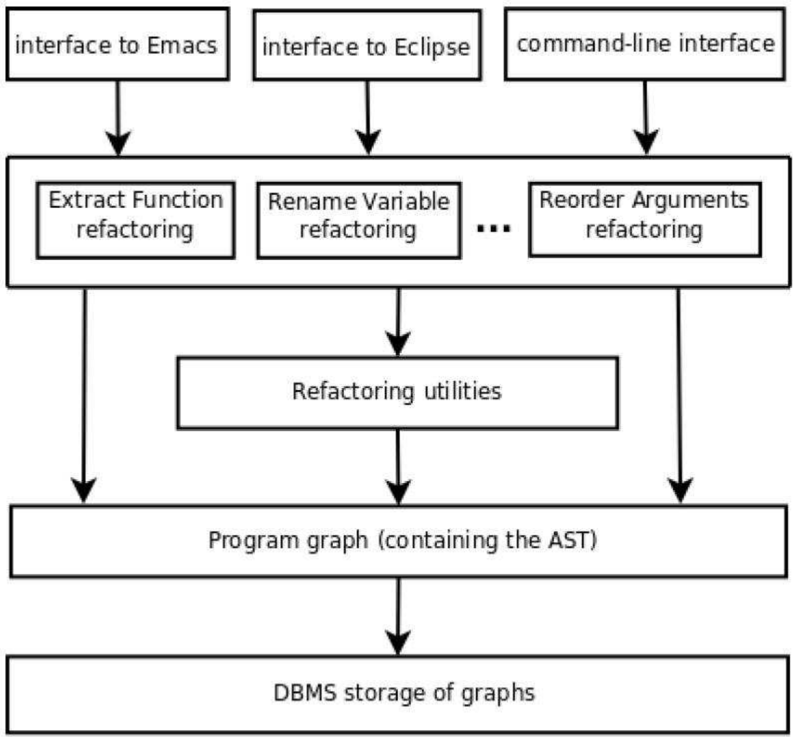
interface to Emacs
interface to Eclipse
command-line interface
Extract Function refactoring
Rename Variable refactoring
...
Reorder Arguments refactoring
Refactoring utilities
Program graph (containing the AST)
DBMS storage of graphs

Figure 1. Logical layers of RefactorErl

schema can be passed to the storage server, which uses this information to increase the efficiency of storing and querying data.

2. The next layer provides consistent Erlang semantical graphs which can be manipulated only through the syntax tree. This is achieved by restricting the interface of the underlying storage layer: the same query operations are available, but manipulations are limited, and more controlled.
3. The next layer maps high level refactoring concepts, basic operations, checks, and transformations to low level graph queries and manipulations.
4. The highest abstraction level layer contains the high level definition of refactorings.
5. The interface layer provides user level access to refactorings.

Semantical analysis is performed by several “analyzer plugins”, which are Erlang modules that concentrate on a specific type of analysis. These modules are used through a callback interface. Semantical consistency is ensured between *semantical transactions*. Graph manipulations start a new transaction, which means that within a transaction consistency is no longer guaranteed, so queries are disabled. The transaction must be explicitly closed. At this point the semantical analyzer modules restore consistency, and queries are enabled again.

6. Skills developed

The RefactorErl project course improves the students' skills in at least four main areas. First, it advances knowledge in the *functional programming paradigm* and in Erlang by requiring that students design and write large, complex programs in this functional language. Secondly, the students must analyze the side conditions of refactoring transformations, hence they get acquainted with static analysis techniques which enhances their understanding of the *semantics* of language constructs [26]. Thirdly, since the parser of a refactoring tool is significantly different from that of a compiler, the students learn how to apply their knowledge in *compiler construction* in flexible, creative ways. Finally, and maybe most importantly, they get experienced in the theory and practice of *software engineering* [34].

During the project work, all major skills needed in software technology are utilized. The students have to understand an existing complex software, comprehend the specification of the task they have to solve, design a program component according to existing interfaces, invent further interfaces, write, maintain, test and document code, and present their work in a collaborative environment.

Furthermore, the students have to use tools for teamwork (versioning system, project management tracking system, wiki, mailing list), for documentation (edoc, LaTeX) and for testing (QuickCheck, CruiseControl) [12]. They often have to develop their own application specific test tools. Finally, they learn how to apply a refactoring tool in the software engineering process.

A major difference of an industrial project compared to university exercises is the amount of collective and individual responsibility in maintaining high quality and respecting deadlines. On the other hand, the fact that the results of the project are used in the software industry (both by Ericsson directly, and by the Erlang community) is highly motivating.

The project is about producing a refactoring tool for Erlang. This involves the development of a refactoring-specific parser (coping with the presence of preprocessor), and semantic analyzer modules for static analysis. Experiments have to be carried out on the syntactic and semantic coverage of Erlang through case studies and test suits. These experiments should point out those uses of the Erlang language which are important in industrial applications and identify the extreme uses with no practical significance. To achieve this, understanding and analyzing huge bodies of (confidential) industrial legacy code from the point of view of refactoring is necessary. Furthermore, the industrial code repository is also used for measuring the efficiency and the responsiveness of the developed refactoring tool.

All of the above tasks require research activities, such as reading literature, algorithm design and analysis etc. Master students can obtain skills useful in an academic career. They can continue their work as a PhD student ensuring the continuity of knowledge transfer.

7. Results

Each student participating in the project understands the RefactorErl system and solves a well-defined subproblem by writing 1000–3000 lines of quality code. Such subtasks solved by students were e.g. the database back-end [31], the first parser, concrete refactorings (such as Extract Function [35], Inline Function [5], Move Function [13], Tuple Function Arguments [29], Renaming Variable and Function), clustering modules and functions, defining and implementing appropriate fitness functions [22], the analysis and evaluation of module structure, and an algorithm for splitting modules [25].

The RefactorErl project is not a typical project in the sense that it addresses non-trivial problems of semantics of programming languages. The software being developed shows many of the features of today’s applications, worth for the students to learn: model based architecture, databases, user interface, distributed middle tier, XML etc. However, the project also requires that the students solve interesting problems about efficient data representation and algorithms. Even PhD students can find appropriate research topics in this project, since the static analysis of a dynamically typed language with concurrent and distributed facilities is really challenging. The choice of the project topic is the key factor for allowing a team structure in which a large number of PhD and master students can cooperate. This team structure, on the other hand, ensures the continuous knowledge transfer. Another important factor is that the industrial partner (Ericsson Hungary) permits the release of the product with an open-source licence and the publication of the research results.

The research components of the project make it possible to present the results in conference and journal papers, PhD and MSc theses, and to provide special education environment for the most talented students (e.g. in the Scientific Students' Association [29], [6], [12]). Our students achieved good results at student research competitions, and were successful in related projects at our partners in Kent, Sheffield and London [23], [12], [24], [28].

Ericsson Hungary was able to use our tools for restructuring a large, industrial code base [25].

8. Discussion

Project courses are often part of undergraduate curricula in software engineering education. Many problems arise during the design and management of such courses. The literature on this topic is quite broad, therefore we point out here only some interesting issues.

Grundy [11] describes the design space for project courses (industrial or artificial topic, individual or team work, self-selected or lecturer organized teams, the length of the project, management, assessment, supporting courses) and evaluates some example courses. In this design space the situation of our project course is the following. Different teams of students work on different industrial projects. The students may choose whether they take this course, and which team they wish to belong to. Most of the students work at least three semesters on the projects. The teams are managed by staff members and PhD students. Staff members assess students by quality of process, and industry clients assess the team by quality of product. The project course needs no supporting lectures, since the students complete SE courses beforehand.

Many undergraduate-level project courses have to deal with a large number of students, which requires significant human resources from the faculty staff. Knight and Horton [19] emphasises the need for a master/apprentice relationship between teacher and student, which is hard to achieve if there are too many projects and too many students. They propose “studio labs” where some randomly selected student groups make presentations to the entire class about their recent achievements and their progress. Evaluation of these presentations by faculty (and by class members) reveal the majority of mistakes made by the class without having to review the work of the entire class. In the project course described in this paper, however, studio labs are not needed. Each project involves PhD students, who are able to review the work of the undergraduate and master students, and lead the technical discussions about design and implementation decisions. The involvement of PhD students is made possible by selecting the project topics in such a way that they require innovation and PhD-level research as well as software development.

Keen, Lockwood and Lamp [18] describes a project course which is more focussed on teaching human interaction than similar courses. They introduce the team-of-teams approach to advance towards this goal: they build a hierarchical team with small sub-teams, project managers and domain experts. The two-level team structure is not required in our projects, and the role of project managers and experts is assigned to faculty members and PhD students.

Michaelsen [27] discusses the basic criteria for successful team-based learning. His Principle 2 (Students Must be Made Accountable) turned out to be critical in our projects as well. On the one hand, the tasks assigned to students have to be selected properly, and this directly affects the structure of the software developed in the projects. On the other hand, the frequent and rather long project meetings are necessary for monitoring the progress and evaluating the (quantity and quality) of work of each student. Our meetings also amply address Principle 4 (Students Receive Frequent and Immediate Feedback).

Page [32] reports on a team-based project course using ACL2. His results are related to ours in two ways. Firstly, he finds that functional programming is suitable, if not preferable for software engineering. Secondly, besides solving practical SE tasks, his students succeed to learn to cope with theoretically challenging problems, such as the development of software verified with mechanical logic.

Turcsányi-Szabó [36] describes the TeaM Lab (Teaching with Multimedia) activities that add up from course project works, undergraduate diploma works, PhD dissertations of students, as well as all local and international projects that TeaM Lab participates in. These ICT projects, similarly to ours, are long-running and provide results used in the real life: they are directly utilized in public education. However, these projects are for informatics teacher training, and hence they are methodology oriented, with less software engineering focus. Due to the nature of the results’ potential capitalization, the projects are supported by research grants rather than industrial partners.

The European University Association (EUA) fosters the collaboration of universities and industrial partners in doctoral education [4]. Added values of the collaboration are enhanced knowledge exchange, research with academic standards providing strategic values for industry, broadening employability of doctorate holders, and reinforcement of university-business cooperation. Our project course clearly contributes to the Open Innovation Model, and extends the goals of EUA by preparing graduate students for the participation in collaborative doctoral education.

To conclude, our course is organized around projects that are running for several years with students regularly joining and leaving the teams. The tasks for the students have to be selected in a way that tolerates the fluctuation in the team size and structure. The choice of the project topics is the key factor for allowing teams in which a large number of PhD and master students can cooperate.

References

[1] J. Armstrong, R. Virding, M. Williams and C. Wikstrom, *Concurrent Programming in Erlang*, Prentice Hall, 1996.

[2] P. Bauer, *Designing an XML model for the HypereiDoc framework*, National Scientific Students' Association Conference, Debrecen, Hungary.

[3] P. Bauer, Zs. Hernáth, Z. Horváth, Gy. Mayer, Zs. Parragi, Z. Porkoláb and Zs. Sz. Sztupák, *HypereiDoc – an XML based framework for supporting cooperative text editions, Advances in Databases and Information Systems (ADBIS 2008)*, Vol. 5207, Lecture Notes in Computer Science (ISSN 0302-9743), Springer, 2008, 14–29.

[4] L. Borrell-Damian, *Collaborative Doctoral Education, University-Industry Partnerships for Enhancing Knowledge Exchange*, DOC-CAREERS Project, European University Association, 2009.

[5] I. Bozó, *Erlang Refactoring: Inline Function*, BSc. thesis, Eötvös Loránd University, Budapest, Hungary, 2008.

[6] I. Bozó and M. Tóth, *Function-oriented Refactorings in Functional Languages*, 1st prize at National Scientific Students' Association Conference, Debrecen, Hungary, 2009.

[7] P. Diviánszky et al., *Infrastructure for Analysis of F# Programs*, Poster, ELTE Innovation Day, 2009.

[8] Eötvös Loránd University, Domain specific language for digital signal processing, `http://dsl4dsp.inf.elte.hu/`.

[9] Eötvös Loránd University, HypereiDoc, `http://hypereidoc.elte.hu/`.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[11] J. Grundy, A comparative analysis of design principles for project-based IT courses, *ACSE* (1997), 170–177.

[12] Cs. Hoch, *Testing Erlang with QuickCheck*, National Scientific Students’ Association Conference, Debrecen, Hungary, 2009.

[13] D. Horpácsi., *Erlang refactoring: Move functions between modules*, BSc. thesis, Eötvös Loránd University, Budapest, Hungary, 2008.

[14] Z. Horváth (Ed.), *Central European Functional Programming School*, Vol. 4164, First Summer School, CEFP 2005, Budapest, Hungary, July 2005, Revised Selected Lectures, ISBN 3-540-46843-9, Lecture Notes in Computer Science (ISSN 0302-9743), Springer, 2006, 256 pages.

[15] Z. Horváth et al., Refactoring Erlang Programs, `http://plc.inf.elte.hu/erlang/`.

[16] Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, T. Nagy, M. Tóth, A. Víg and R. Király, *Building a Refactoring Tool for Erlang*, Int’l Workshop on Advanced Software Development Tools and Techniques, ECOOP 2008, Paphos, Cyprus, July 8, 2008, `http://smallwiki.unibe.ch/wasdett2008/`.

[17] Z. Horváth, R. Plasmeijer, A. Soós and V. Zsók (Eds.), *Central European Functional Programming School*, Vol. 5161, Second Summer School, CEFP 2007. Revised Selected Lectures, Lecture Notes in Computer Science, (ISSN 0302-9743), Springer, 2008, 301 pages.

[18] C. Keen, C. Lockwood and J. Lamp, *A client-focused, team-of-teams approach to software development projects*, Software Engineering: Education & Practice, 1998. International Conference, 26–29 Jan 1998, 34–41.

[19] J. C. Knight and T. B. Horton, *Evaluating A Software Engineering Project Course Model Based On Studio Presentations*, 35th ASEE/IEEE Frontiers in Education Conference, 2005.

[20] M. Kovács and Zs. Sz. Sztupák, *Layered XML files*, 2nd prize at National Scientific Students’ Association Conference, Debrecen, Hungary, 2009.

[21] T. Kozsik, Z. Csörnyei, Z. Horváth, R. Király, R. Kitlei, L. Lövei, T. Nagy, M. Tóth and A. Víg, *Use Cases for Refactoring in Erlang*, Central European Functional Programming School (The Second Central European Summer School, CEFP 2007, Cluj, Romania, June 23–30, 2007), Revised Selected Lectures, LNCS 5161, Springer Verlag, 2008, 250–285.

[22] H. Köllő, *Refactoring Erlang Programs: Clustering Modules*, BSc. project, Eötvös Loránd University, Budapest, Hungary, 2008.

[23] H. Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg and T. Nagy, *Refactoring Erlang programs*, The Proceedings of 12th International Erlang/OTP User Conference, Stockholm, Sweden, 2006, 10 pages, `http://www.erlang.se/euc/06/`.

[24] H. Li, S. Thompson, Gy. Orosz and M. Tóth, *Refactoring with Wrangler, updated*, Proceedings of the 2008 SIGPLAN Erlang Workshop, ISBN: 978-1-60558-065-4, ACM, 2008, 61–72.

[25] L. Lövei, Cs. Hoch, D. Horpácsi, H. Köllő, T. Nagy and A. Víg, *Refactoring Module Structure*, Proceedings of the 2008 SIGPLAN Erlang Workshop, ISBN: 978-1-60558-065-4, ACM, 2008, 83–89.

[26] L. Lövei, Z. Horváth, T. Kozsik, R. Király and R. Kitlei, *Static rules for variable scoping in Erlang*, Vol. 2, Proceedings of the 7th International Conference on Applied Informatics, 2008, 137–145.

[27] L. K. Michaelsen, Getting Started with Team Based Learning, *Team-Based Learning: A Transformative Use of Small Groups*, Greenwood Publishing Group, 2002, 27–52.

[28] T. Nagy and A. Víg, *Erlang Testing and Tools Survey*, Proceedings of the 2008 SIGPLAN Erlang Workshop, ISBN: 978-1-60558-065-4, ACM, 2008, 21–28.

[29] T. Nagy and A. Víg, *An Erlang refactor step: Tuple function arguments*, 2nd prize at National Scientific Students’ Association Conference, Miskolc, Hungary, 2007.

[30] T. Nagy and A. Víg, *Erlang refactor tool*, Master thesis, Eötvös Loránd University, Budapest, Hungary, 2007.

[31] T. Nagy and A. Víg, *Storing Erlang source code in database*, BSc. thesis, Eötvös Loránd University, Budapest, Hungary, 2006.

[32] R. Page, Engineering software correctness, *Journal of Functional Programming* **17**, no. 6 (November 2007), 675–686.

[33] Zs. Parragi and Zs. Sz. Sztupák, *Digital representation, presentation and collaborative edition of philological texts*, 3rd prize at National Scientific Students’ Association Conference, Debrecen, Hungary, 2009.

[34] I. Sommerville, *Software Engineering*, (7th Edition) (International Computer Science Series), Addison Wesley, 2004.

[35] M. Tóth, *Erlang Refactoring: Extract Functions*, BSc. thesis, Eötvös Loránd University, Budapest, Hungary, 2008.

[36] M. Turcsányi-Szabó, *Blending projects serving public education into teacher training*, Education for the 21st Century - Impact of ICT and Digital Resources, IFIP 19th World Computer Congress, TC-3 Education, IFIP series Vol. 210, Springer, 2006, 235–244.

[37] Ceepus Network CII–HU–19 International Cooperation in Computer Science, `http://aszt.inf.elte.hu/~ceepush81/`.

ZOLTÁN HORVÁTH, TAMÁS KOZSIK and LÁSZLÓ LÖVEI
DEPARTMENT OF PROGRAMMING LANGUAGES AND COMPILERS
EÖTVÖS LORÁND UNIVERSITY
BUDAPEST
HUNGARY

E-mail: `hz@inf.elte.hu`

E-mail: `kto@inf.elte.hu`

E-mail: `lovei@inf.elte.hu`

(Received October, 2009)