DEVELOPMENT PROCESS FRAMEWORK FOR SOFTWARE BASED ON OPEN-SOURCE COMPONENTS: KEY CONCEPTS

JAKUB SWACHA, KAROLINA MUSZYŃSKA, ZYGMUNT DRĄŻEK

Summary

The open source community produces a wide range of software products every year. If a problem is typical, the probability is high that there is an open-source software product already available that solves it. However, if a set of functionalities and/or non-functional requirements demanded by end-users is carefully compared to what the available software offers, significant differences often appear. This means it is not a routine case that an open source system can simply be adopted; it has to be adapted frequently. In this paper, we propose a new process framework for developing software based on open-source components. We call it FEChADO, which is an acronym for the six steps it consists of: finding available solutions, evaluating solutions from the list, choosing the most appropriate solution, adapting the solution, developing new modules and obtaining users' feedback. The framework is a direct result of these authors' practical experiences from developing software based on open-source components.

This paper is an extended version of the poster -to be presented at the 14th International Conference on Enterprise Information Systems in Wroclaw, Poland [14].

Keywords: open source software, software adaptation, development methodology

1. Introduction

1.1. Motivation

The open-source software becomes increasingly popular every year [8]. Open-source solutions can be found for many typical business applications.

However, ready-made, open-source solutions rarely fully meet the end users' requirements. Therefore, most of them require some level of adaptation in order to make the users satisfied.

Although there is vast literature on the software development process and significant literature on selecting open-source software (to be discussed in section 2), until now little has been written on the process of developing software based on open-source components, where the main effort lies in choosing the most appropriate existing solutions first, rather than developing new ones; but then, whichever of them is chosen, there is the non-trivial adaptation stage planned that adapts the chosen software to the requirements of the specific end users.

1.2. Problem setting

We understand adapting open-source software as tailoring it to the needs of a specific end-user or a group of them. Such modifications are rarely useful for other users, so usually there is no reason to propose them as improvements for a future release or create a branch of the software to which they are applied.

The permission for users to modify code for private purposes is a core property of the open-source software. Thus, as there is usually no reason for further redistribution of the modified software, the relevant license requirements do not matter.

Still, in case that the changes made to the code are more than cosmetic; for instance, a new component is developed, contributing it to the open source community should be considered, provided both the license requirements and the organization's internal regulations on software developed by its employees permit such thing.

1.3. Approach

We assume open-source software adaptation to be a repeatable process, hence a framework can be defined for its efficient execution. Such a framework could be based on theoretical inference, starting with more or less abstract requirements, or obtained by generalization of practical experiences with implementation of the aforementioned process. We chose the latter and use observations made during our involvement in open-source software adaptation, especially the experiences gained within the BalticMuseums 2.0 and BalticMuseums 2.0 Plus projects (see e.g. [15]).

We consider six main stages that form the development process framework:

1) Find available solutions,
2) Evaluate solutions from the list,
3) Choose the most appropriate solution,
4) Adapt the solution,
5) Develop new modules,
6) Obtain users' feedback.

For some of the stages, typical phases are defined. For all of the stages, an exemplary method of implementation is described. The proposed framework can be seen as high-level and flexible as it does not enforce a specific method for any of the stages; any method is suitable for use provided it accepts the available inputs (initial, or obtained from the preceding stage) and produces the required outputs (final, or needed by the subsequent stage).

1.4. Contributions

Our main contribution is the development process framework that can be used in various organizations for adapting open-source software.

We call it FEChADO, which is an acronym of the six steps it consists of.

2. Related work

Before formulating the proposed process framework numerous research findings, existing solutions and scientific papers were verified and examined.

An interesting example of a method for the qualification and selection of open-source software, called QSOS, is described by Atos Origin [1]. The QSOS method assumes the realization of four stages leading to selection of the most appropriate open-source solution. The stages include: definition of frames of reference, evaluation of software by identifying its main characteristics, defining its functional coverage and risks from the user's and service provider's perspective, qualification of the software using filters to translate needs and constraints to the selection of most suitable solution, and the selection of software based on the outputs of the previous stages.

M. Cabano, C. Monti and G. Piancastelli based their context-dependent evaluation methodology [2] on the common structural pattern shared by the most widely-known evaluation models such as the Open-Source Maturity Model created by B. Golden [7], and another one created by the CapGemini consulting company [5] or Business Readiness Rating for Open Source created by Carnegie Mellon West and Intel [11]. This common pattern for solving the open-source software evaluation problem consists of three general phases: data gathering, data analysis and numerical synthesis. The assessment process of the context-dependent methodology consists of three phases: context analysis, which defines the necessities and requirements, preliminary selection, which addresses the most critical metrics and the filtered selection, which estimates the remaining products by a complete set of metrics [2].

Comparison of QSOS and OpenBRR open-source software evaluation methods available in work [4] indicates the advantages and weaknesses of both approaches, which gave these authors additional insight into the FEChADO framework.

The need for appropriate open-source software identification and selection is also described in relation to the assembly methodology, which, similarly to other agile methodologies, is an iterative process for application development, using frequent integrations and based on putting together existing software components such as SOA services and open-source software. The process assumes identifying, selecting and integrating the needed services and open-source software and building a new user interface for the thus created new application [10].

Some valuable information concerning ways of searching for and the evaluation of open-source software comes also from practitioners like R. Galoppini, who, in his pragmatic methodology, indicates the best sources to find the desired open-source software and lists the evaluation criteria to be applied, which include: code maturity, project popularity, case study availability, books, community size, commercial support, training, documentation, bugs reactivity, source, license, modifiability, roadmap and sponsorship [6].

The open-source software evaluation process described by D. A. Wheeler [17] consists of four steps: identify candidates, read existing reviews, compare the leading programs' attributes to the needs and analyse the top candidates in more depth. Wheeler also suggests a set of important attributes for comparing candidate open-source software and additionally an in-depth analysis for adding functions and the analysis of software security for the top candidates.

A slightly different approach, which concentrates on the software quality is described in a paper by G. Polancic, R. V. Horvat and T. Rozman [12]. The proposed model for evaluating an open-source solution is based on the multiple criteria of software quality, which uses easy, accessible quantitative data.

An interesting evaluation process based on a survey of six open-source software evaluation approaches and tested on real software systems is presented in the master's thesis by Karin van den Berg [16]. The elaboration lists the evaluation criteria, explains in detail why they are important and proposes the selection and evaluation methods. The main goal of the abovementioned open-source software evaluation methods and frameworks is to identify, assess, sometimes also compare and select open-source products. They differ in the number of phases or evaluation criteria, but the goal remains the same. The proposed FEChADO framework concentrates on the software development process based on existing open-source software components; it assumes that the open-source software evaluation process must be followed by the additional adaptation of the selected solution, which can include the development of new modules and the application of users' remarks.

3. FEChADO framework

We call our process framework 'FEChADO', which is an acronym of the six steps it consists of (see Fig. 1).

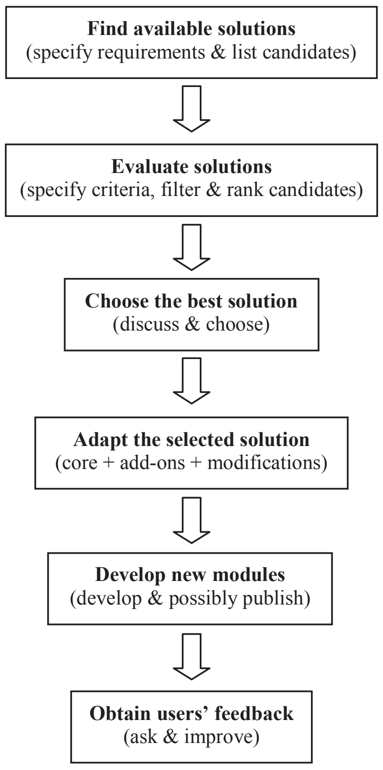
Find available solutions
(specify requirements & list candidates)
Evaluate solutions
(specify criteria, filter & rank candidates)
Choose the best solution
(discuss & choose)
Adapt the selected solution
(core + add-ons + modifications)
Develop new modules
(develop & possibly publish)
Obtain users' feedback
(ask & improve)

Fig. 1. Overview of the FEChADO process framework

Stage 1: Find available solutions.

The output of the first stage is an exhaustive list of software that should be considered for adoption.

Phase 1.1: Specifying requirements. In the first step, the set of functionalities and non-functional requirements of the end-users has to be defined.

If the general description of the software fits into an established software category, a list of suggested requirements may be composed, based on characteristics of the software belonging to that category. Otherwise, a brainstorming session involving both designers and users can be organized to obtain such list.

This list is then used as the basis for a questionnaire to be filled in by the end users. The basic type of question they answer is how they value each proposal (don't want, don't need, could use, should be, must be). They are also asked for their own proposals that were not mentioned in the questionnaire.

One may ask end users more detailed questions, such as how they envision specific features to be implemented.

An extended questionnaire containing technical questions (e.g., regarding available or possible to obtain hardware and software platforms and their parameters) could be prepared for IT administrators.

The results from the questionnaires are then processed and analysed in order to obtain a list of requirements arranged in three groups:

1) core requirements (that must be met),
2) additional requirements (that should be met),
3) special requirements (that only specific types of end users believe should be met).

The list is then presented to the end users for possible correction and re-evaluation.

Phase 1.2: Listing available solutions. There are various sources that can be searched for solutions that possibly match the specified requirements (see e.g., [7]). We recommend starting with searching open-source project portals, such as sourceforge.net and freecode.com, as well as using web search engines. Multiple search phrases should be tried, as different words may be used to describe equivalent functionalities.

The names of programs found first may be searched for to find competing solutions or even feature comparison tables that will be very helpful in the evaluation stage. If the software fits into an established software category, the latter can be found on Wikipedia (see e.g., [9]) or specialized sites (see e.g., [3]).

Note that the search should be comprehensive, i.e. continued until no more matching solutions can be found. It is better to include candidates who will be deleted from the list during the preliminary evaluation rather than to omit a possibly optimal solution.

Stage 2: Evaluate found solutions

The evaluation stage consists of four phases. The first three involve only the developers, the fourth one also the end users.

Phase 2.1: Specifying evaluation criteria. The evaluation criteria are specified by assigning measurable fulfilment levels to the requirements. Every criterion should have at least one defined fulfilment level: acceptable; that is, solutions evaluated below this level must not be chosen. However, there could be more levels defined so that solutions that pass the acceptable level could be compared between each other. A satisfactory level can be defined, such that no solution can be evaluated as better with regards to a given criterion if they both attain this level. There may be an unlimited number of intermediate levels between acceptable and satisfactory levels.

Note that for extensible systems, the acceptable level should be described as a level that can be implemented with available resources and is not already implemented in a sub-optimal way.

The evaluation criteria can be arranged, depending on their measurability, into the following groups:

1) objective and easily measurable,
2) objective and not easily measurable,
3) subjective.

Although equal weights may be given to the evaluation criteria, usually some criteria are more important than others. The weights may be established using a simple ranking done by end users, or using more sophisticated approaches, like pair-wise comparisons (see e.g., the AHP method [13]). Notice that special requirements should only be weighted by the specific types of end users who are interested in them.

Phase 2.2: Preliminary evaluation. The goal of the preliminary evaluation is to shorten the candidate list by removing software that does not attain an acceptable level for criteria based on the core requirements. In this phase, only objective and easily measurable criteria, i.e., those that could be checked based only on product fact sheets without the need for installing/testing demo versions or careful reading of documentation, should be considered. Note that the solutions removed from the candidate list after the primary evaluation will not be considered later. It is therefore important at this stage to evaluate only those criteria that actually can be checked without testing the software. This remark is especially valid for modular software whose description may not list functionalities that are available as third-party add-ons.

Phase 2.3: Main evaluation. The goal of the main evaluation is to produce a short list of solutions (or sets of solutions) that will be considered for the choice. This phase permits much longer evaluation time per candidate solution than the preliminary evaluation; therefore, both easily and not easily measurable criteria are now checked, i.e., also those that require installing/testing demo versions and/or careful reading of documentation. Each candidate is supposed to be evaluated by an expert, and only objective criteria should be considered at this stage.

First, as in the preliminary evaluation, the list is trimmed by eliminating software that does not attain an acceptable level for the objective criteria (now also not easily measurable) based on the core requirements.

Both the criteria fulfilment levels and criteria weights are then scaled and normalized, so that only values from a certain range (e.g., <0; 1>) are obtained. Thanks to that, an aggregate measure for each candidate solution and group of criteria can be constructed by summing up products of criterion fulfilment level and criterion weight for all criteria belonging to that group. Thus, for every candidate solution three aggregate measures are calculated based on: only core criteria, combined core and additional criteria, and all criteria (i.e., including the special ones).

Next, three ordered lists are produced, each by sorting the input list by one aggregate measure. Subsequently, a single combined list is obtained by taking a specified number of candidate solutions from the top positions of each ordered list. A suggested procedure is to take the first item from the first list, then the first item from the second list, provided it is not on the output list already, and continue with subsequent items until a preferred number of solutions for in-depth evaluation is obtained. It is suggested that between three to five solutions should be qualified for the next phase.

Phase 2.4: In-depth evaluation. The goal of the in-depth evaluation is to prepare a full set of criteria fulfilment information for each candidate solution accepted for the short list that will be considered for making the choice. The evaluation procedure at this phase may be complex and include subjective criteria. Every candidate is supposed to be evaluated by several people, working as a team or independently – the evaluation results are averaged in the second case. In contrast to the two previous phases, the in-depth evaluation should be performed not only by experts or developers, but also by representative end users, so that their opinion on the usability and general look could be learned.

This phase is considered finished when all criteria are measured for each candidate on the list. If some criterion still could not be measured, it need not be removed from the criteria set, rather an explanation should be appended to the results report.

Stage 3: Choose the most appropriate solution

The goal of stage 3 is to select a single solution (a software system or a set of such) that will be adapted and then adopted.

Phase 3.1: Discussion of evaluation results. A meeting should be organised to discuss the evaluation results. The following stakeholders should take part in it: sponsors of the project (actual decision makers), members of the development team, external experts having knowledge of the software on the short list – especially if there are no highly qualified individuals in a specific software element within the development team, and representatives of the end users – especially those who took part in the in-depth evaluation phase.

The meeting should start with a presentation of the evaluation results. Then, during the discussion, every person taking part in the meeting should be allowed to express his or her opinion on the preferred software. Moreover, members of the respective groups should provide additional information that could impact the choice, e.g.:

1) the end users who participated in the in-depth evaluation should point to the drawbacks or special advantages of respective solutions;
2) the invited experts should clarify, if the mentioned drawbacks/special advantages are actually specific to the respective solutions, or are just the result of specific configuration or usage;
3) the members of the development team should declare if they are capable of fixing the mentioned drawbacks or adding similar advantages to other candidate solutions and what would be the cost in resources of doing that;
4) the sponsors of the project should declare if they are ready to contribute resources necessary for the discussed changes;
5) the end users should state if they would accept a solution with the discussed drawbacks or without discussed advantages.

Phase 3.2: Making the choice. The decision on choosing the solution for adoption is made by the sponsors of the project. It should be based on the results of the evaluation process, but if two or more solutions were evaluated closely, the decision maker(s) should pick one of them using their own opinion rather than a tiny difference in the aggregated measure values.

Stage 4: Adapt the solution

Adapting the solution consists of:

1) acquiring, installing, and configuring the core solution,
2) acquiring, installing, and configuring the required add-ons,
3) applying modifications required for the solution to meet the requirements.

The extent of modifications may be various. Sometimes, it will be just a few lines of code inserted in a single file and sometimes thousands of lines spanning through multiple modules. What makes the modifications applied at this stage distinctive from those of the next stage is that they are aimed at the requirements of the specific end user group, and as such they will rarely be useful for other users and that they are not usually defined as a separate entity (module or even function), being often a list of file/line updates.

It is very important that every modification made to the original solution is well explained in the technical documentation of the final product, in terms of its purpose, relation to other modifications, assumed conditions and possible risk factors. Much effort should be spent on regression testing to assure that the modification does not hurt the stability and/or security of the solution. Its impact on performance also has to be tested, and negative results should initiate an attempt to optimize the relevant code.

The modifications are applied to a certain version of the solution, installed at the time of development. It may raise a number of problems whenever the solution is updated. First of all, the modifications may have to be reapplied. Therefore, there must be a list of updates to the core solution files that have to be applied after each update, or, preferably, a script that automates the process.

Secondly, the updated version of the solution may render the modifications non-applicable in their original form. In such case, either the modifications have to be re-implemented, or the used solution version must be frozen. The latter should be only a temporary solution in case there are important security-related patches released.

If applying the modifications was automated with a script, it should also check the context and produce an error message in case the modification could not be applied.

Stage 5: Develop new modules

Sometimes, the modifications related to a certain group of requirements can be implemented as a new module for the chosen solution. Whenever this is possible, it should be the preferred way of implementing modifications, because they become easier to manage, and, as they are less harmful to the existing code, they are also less prone to compatibility problems in the event of future patches to the core solution.

Developing a module should be done by stringently following the guidelines defined in the solution's documentation. The new module should integrate seamlessly with the solution and be configurable, preferably from the solution's administration panel, if such exists.

The developed module may contain a set of functionalities that could be interesting for other users. If the organization's internal regulations permit the distribution of software developed by its employees as open source and there are no redistribution constraints in the license of the original solution that could be violated, it should be considered to contribute the module to the open-source community.

In case it is decided to publish the new module as open source, it should be trimmed from elements that are part of the adaptation of the solution, that have any value only for the sponsor's organization. Depending on the number and character of such elements, it may be accomplished in three ways:

1) by turning the adaptation elements into a profile of module configuration settings,
2) by developing additional module, only for internal usage,
3) by branching the module into internal (full) and external (limited) versions.

If the module published as open source gains popularity, it may even attract external developers who may improve it. In such case, it should be checked if the improved version of the module could be used instead of the one developed internally. If there are noticeable advantages and the risk of ensuing problems is negligible, the replacement should be made.

Stage 6: Obtain users' feedback

It is important that the end users' opinions about the final product are gathered. The opinions that pertain to the implemented modifications can be used by the internal development team to improve them. Contrasting opinions should be resolved via discussion with the involved users.

The opinions that refer to the chosen solution should be passed to its original developers, especially bug reports and feature requests.

In order to facilitate the feedback process, a web form should be made available for the end users, so that their opinions could be reported easily.

4. Practical experiences

The development process framework outlined in this paper has been applied in a complex project [15], consisting of five components, four of which were adaptations of open-source systems (system-level components, including web server and database management system, which are also open-source, are not counted here as they were not subject to the proposed framework) and the fifth one was developed from scratch.

The core of the framework was created after developing the first component of the mentioned project, and applied, in limited or full extent, to the remaining ones. The development of the framework was evolutionary: each application resulted in experiences that allowed for further improvement of the framework.

The framework has been evaluated positively by the respective decision makers who especially liked its non-intrusive guidance in selecting the right software solutions as well as developers who particularly liked its suggestion of a development path without introducing unnecessary complexity to the development process.

A number of improvement suggestions was collected and mostly incorporated in the framework, though their description is too detailed to be discussed in this paper.

5. Conclusions

We have described key concepts of a new framework for a software development process based on open-source components. The framework introduced in this paper can be applied to any software development process that involves open-source software adaptation. It helps achieve quality of the final products at the same time being simple and non-obtrusive.

The framework has been applied in practice and earned positive opinions from both sponsors and developers involved in the project.

The outline of the framework presented in this paper is enough to mimic its stages and phases in software development processes. A work is undergoing, aimed at publishing an extensive description of the framework as a book.

Bibliography

[1] Atos Origin: *Method for Qualification and Selection of Open-Source software (QSOS), version 1.6,* 2006, http://www.qsos.org/download/qsos-1.6-en.pdf [Accessed Feb. 2012].

[2] Cabano, M., Monti, C., Piancastelli, G.: Context-Dependent Evaluation Methodology for Open Source Software, [in:] *Open Source Development, Adoption and Innovation* (eds. J. Feller, B. Fitzgerald, W. Scacchi, A. Sillitti), Springer, New York, 2007, pp. 301–306.

[3] *Compare Content Management Systems*, 2012, http://www.cmsmatrix.org [Accessed Feb. 2012].

[4] Deprez, J. C., Alexandre, S.: Comparing Assessment Methodologies for Free/Open Source Software: OpenBRR & QSOS, [in:] *Product-Focused Software Improvement. 9th International Conference, PROFES 2008. LNCS 5089* (eds. A. Jedlitschka, O. Salo), Springer, New York, 2008, pp. 189–203.

[5] Duijnhouwer, F. W., Widdows, C.: *Capgemini Expert Letter – Open Source Maturity Model,* 2003, http://bolsa.info.unlp.edu.ar/campamento/campamento/documentos/GB_Expert_Letter_Open_Source_Maturity_Model_1.5.3.pdf [Accessed Feb. 2012].

[6] Galoppini, R.: *How to find an Open-source Alternative to Commercial Software*, 2011, http://www.masternewmedia.org/open-source-software-tools-and-directories-where-to-find-them-how-to-evaluate-them/ [Accessed Feb. 2012].

[7] Golden, B.: *Succeeding with Open Source*, Addison-Wesley Pearson Education, Boston, 2005.

[8] Hauge, Ø., Sørensen, C.-F. and Conradi, R.: Adoption of Open Source in the Software Industry, [in:] *Open Source Development, Communities and Quality* (eds.: B. Russo, E. Damiani, S. Hissam, B. Lundell, G. Succi). Springer, Boston, 2008, pp. 211–221.

[9] *List of content management systems*, 2012, http://en.wikipedia.org/wiki/List_of_content_management_systems [Accessed Feb. 2012].

[10] Michelson, A.: *The assembly methodology using SOA and open source software.* SearchSOA, 2012, http://searchsoa.techtarget.com/tip/The-assembly-methodology-using-SOA-and-open-source-software [Accessed Feb. 2012].

[11] OpenBRR: *Business Readiness Rating for Open Source. A Proposed Open Standard to Facilitate Assessment and Adoption of Open-Source Software*, 2005, http://docencia.etsit.urjc.es/moodle/file.php/125/OpenBRR_Whitepaper.pdf. [Accessed Feb. 2012].

[12] Polancic, G., Horvat, R.V., Rozman, T.: Comparative assessment of open-source software using easy accessible data, [in:] *Proceedings of the 26th International Conference on Information Technology Interfaces – ITI 2004* (eds. V. Luzar-Stiffler, V. H. Dobrić), IEEE, Zagreb, 2004, pp. 673–678.

[13] Saaty, T. L.: *The Analytic Hierarchy Process*, McGraw-Hill, New York, 1980.

[14] Swacha, J., Muszyńska, K., Komorowski, T. and Drążek, Z.: An outline of development process framework for software based on open-source components, [in:] *Proceedings of the 14th International Conference on Enterprise Information Systems*, vol. 2 (eds. L. A. Maciaszek, A. Cuzzocrea, J. Cordeiro), SciTePress, Wrocław, 2012, pp. 183–186.

[15] Swacha, J., Muszyńska, K., Komorowski, T. and Drążek, Z.: Development and maintenance of a multi-lingual e-Tourism website on the example of BalticMuseums 2.0 Online Information Platform, [in:] *Information Management*, Gdansk University Press, Gdansk, 2011, pp. 237–246.
[16] van den Berg, K.: *Finding Open options. An Open-source software evaluation model with a case study on Course Management Systems.* Master Thesis, Tilburg University, Tilburg, 2005, http://www.karinvandenberg.nl/Thesis.pdf [Accessed Feb. 2012].
[17] Wheeler, D. A.: *How to Evaluate Open-Source Software / Free Software (OSS/FS) Programs*, 2011, http://www.dwheeler.com/oss_fs_eval.html [Accessed Feb. 2012].

GŁÓWNE KONCEPCJE METODYKI ROZWIJANIA OPROGRAMOWANIA OPARTEGO NA KOMPONENTACH O OTWARTYM KODZIE

Streszczenie

Każdego roku społeczność oprogramowania o otwartym kodzie wytwarza szeroki wachlarz programów. Dla typowych problemów, prawdopodobieństwo tego, że dostepne jest oprogramowanie o otwartym kodzie, które taki problem rozwiązuje, jest wysokie. Jednakże, gdy porówna się zbiór funkcjonalności czy wymagań niefunkcjonalnych, których domagają się użytkownicy końcowi do tego, co faktycznie oferuje dostepne oprogramowanie, często ujawnia się znacząca rozbieżność. Oznacza to, że do rzadkości nie należy sytuacja, w której otwarte oprogramowanie nie tylko musi zostać skonfigurowane, ale także musi zostać zmodyfikowane. W artykule proponuje się nową metodykę rozwijania oprogramowania opartego na komponentach o otwartym kodzie, FEChADO. Jej nazwa jest akronimem od sześciu wyrazów opisujących etapy jej stosowania: wyszukania dostępnych rozwiązań (Find), oceny znalezionych rozwiązań (Evaluate), wybrania najlepszego spośród nich (Choose), dostosowania go do potrzeb (Adapt), rozbudowy o nowe moduły (Develop), pozyskania opinii użytkowników (Obtain). Metodyka stanowi bezpośredni rezultat praktycznych doświadczeń autorów przy rozwijaniu oprogramowania opartego na komponentach o otwartym kodzie. Artykuł stanowi rozszerzenie treści plakatu, który zostanie zaprezentowany na konferencji ICEIS we Wrocławiu [14].

Słowa kluczowe: oprogramowanie o otwartym kodzie, adaptacja oprogramowania, metodyka rozwijania oprogramowania

Jakub Swacha
Karolina Muszyńska
Zygmunt Drążek
Instytut Informatyki w Zarządzaniu
Wydział Nauk Ekonomicznych i Zarządzania
Uniwersytet Szczeciński
ul. Mickiewicza 64, 71-101 Szczecin
e-mail: jakubs@uoo.univ.szczecin.pl