Knowledge-Based Approach to Effectiveness Estimation of Post Object-Oriented Technologies in Software Maintenance

Mykola Tkachuk[1], Konstiantyn Nagornyi[1], Rustam Gamzayev[1]

[1] National Technical University "Kharkiv Polytechnic Institute",
Frunze str., 21, 61002 Kharkiv, Ukraine
{tka@kpi.kharkov.ua , k.nagornyi@gmail.com , rustam.gamzayev@gmail.com}

Abstract. A comprehensive approach to effectiveness's estimation of post object-oriented technologies (POOT) is proposed, which is based on structuring and analyzing of domain-specific knowledge about such interconnected and complex data resources within a software maintenance process as: 1) structural complexity of legacy software systems; 2) dynamic behavior of user's requirements; 3) architecture-centered implementation issues by usage of different POOT. The final estimation values of POOT's effectiveness are defined using fuzzy logic method, which was tested successfully at the maintenance case-study of real-life software application.

Keywords: post object-oriented technology, effectiveness, crosscutting functionality, knowledge-based approach, fuzzy logic.

Key terms: Software Engineering Process, Knowledge Representation, Decision Support, Model, Metric

1 Introduction: Problem, Actuality and Research Objectives

The most part of modern software systems are developed and maintained using object-oriented programming (OOP) [1]. Well-known and important problem to support such applications are often modifications on many their subsystems and development of new components to implement additional business logic due to new user requirements. In order to emphasize this issue we propose to use in this paper the notion "legacy software system" (LSS), similarly to the terms in software reengineering domain (see, e.g. in [2]). Permanent changes in LSS lead to design instability which causes a so-called crosscutting concern problem [3,4]. The OOP actually does not solve this issue, and usage of OOP-tools increases the complexity of an output source code.

During ten last years some post object-oriented technologies (POOT) were elaborated and became intensive development, especially the most known POOT are: aspect-oriented software design (AOSD) [5], feature-oriented software design (FOSD) [6] and context-oriented software development (COSD) [7]. All these POOTs utilize the basic principals of OOP, but in the same time they have additional features, which allow solving the crosscutting problem electively. From the other hand the usage of any POOT for LSS maintenance and reengineering is related to additional time and other efforts in software development. That is why many researchers emphasize the actual need to elaborate appropriate approaches to complex estimation of POOT's effectiveness usage in real-life software projects. It is additionally to mention that within the context of this paper we are talking about the POOTs which are focused on programming techniques exactly, but not about such software management trends as Extreme Programming (XP), Rapid Application Development (RAD), Scrum and some others [8], which also can be characterized as "post object-oriented" approaches.

Taking into account the issues mentioned above, the main objective of the research presented in this paper is to propose the intelligent complex approach to effectiveness's estimation of using POOTs in software maintenance. The rest of this paper is organized in the following way: Section 2 analyses some critical issues in OOP and reflects the phenomena of crosscutting functionality in software maintenance. In Section 3 the existing POOT are analyzed and the results of their comparing are shown with respect to software maintenance problems. In Section 4 we present the knowledge-based approach for effectiveness's estimation of POOT, which is based on structuring and analyzing of domain-specific knowledge about interconnected and complex data resources within a software maintenance framework. Section 5 presents first implementation issues and the results of test-case for the proposed approach. In Section 6 the paper concludes with a short summary and with an outlook on the next steps to be done in the proposed R&D approach.

2 Some Critical Issues in Object-Oriented Programming and Crosscutting Functionality Phenomenon in Software Maintenance

To meet new requirements existing LSS have to be refined with new classes, which must implement their new functionality. Standard OOP toolkit "proposes" to support additional associations between already existent and new program objects, to modify inheritance tree for classes, to implement new or additional design patterns, e.g. the Gang-of-Four (GoF) patterns [9]. Because of permanent modifications on source code and doing software system re-design, developers face with "bottlenecks" of OOP: increase coupling among classes [10]; increase of depth of inheritance tree (DIT) for class hierarchies [11]; modification of design pattern instances [12,13]; emerging lack of modularity in functionality realization [14].

A number of studies investigate problems of OOP mentioned above, and theirs negative influence on LSS maintenance. High dynamic of requirement changes and these critical issues of OOP induce and propagate an additional development problem: this is a crosscutting concern's phenomena. Crosscutting concern (hereby referred as

"crosscutting functionality" - CF) is a concern emerges on user requirements level and often crosscuts on design level, this is a part of a business logic, which can not be localized in the separate module on source code view but stays separate on requirement view [15]. In literature exists a lot of researches related to CF's properties, multiple patterns of CF and it's interaction with the source code of non-crosscutting functionality, and it's further propagation in system's source code (see e.g. in [13 - 16]). There are some widespread examples of software system features which could be consider as CF: exception management, logging, transaction management, data validation [17]. Nevertheless our own experience in software development and LSS maintenance exposes that almost any system feature, emerged by requirements, on source code perspective could be transformed into CF.

CF has two main properties [18]: scattering and tangling. CF's source code *scatters* among classes (components) of non-crosscutting functions, this happens because of mismatch on end user requirement's level of abstraction, and final realization of this requirement as a feature on the source code level. CF's source code *tangles* (mixes up) with source code of the other functionality, no matter crosscutting or non-crosscutting. Moreover CF could be divided into several types [19]: homogeneous and heterogeneous. *Homogeneous* CF represents the same piece of source code which crosscuts multiple locations in multiple OOP-classes of a software system. *Heterogeneous* CF represents each time unique piece of source code which crosscuts multiple locations in multiple OOP-classes of a software system (see Fig.1).

```
public class Line {
private Point p1, p2;
 Point getP1(){ return
p1; }
 Point getP2() { return
p2; }
 void setP1(Point p1){
 this.p1 = p1;
Display.update(this);}}
public class Oval {
void setPosition(Point
p2){
  this.p2 = p2;
Display.update(this);
}
}
```

// Homogeneous CF

```
public class
CreditCardProcWithLogging{
Logger _logger;
public void debit(CreditCard
card, Money amount)throws
InvalidCardException,
NotEnoughAmountException,
CardExpiredException {
 _logger.log("Starting
debiting"
 + "Card: " + card
 + " Amount: " + amount);
// Debiting
 _logger.log("Debiting
finished"
 + "Card: " + card); }
```

// Heterogeneous CF

Fig. 1. Crosscutting functionality types

As a result, a presence of the CF in software system increases a complexity of the maintenance process [20]:

- CF complicates traceability of various software design artifacts, e.g requirements traceability [21];
- CF decreases understandability of a source code and functionality it realizes;
- source code of LSS becomes redundant;

- Almost impossible to reuse CF solutions, because of lack of modularity.

A conceptual approach, which allows to deal with CF, is a separation of concerns (SoC) [22]. It envisages a *decomposition* and further non-invasive *composition* of CF source code with the rest code of LSS. Decomposition mechanism allows to split source code into fragments and to organize them into easy-to-handle CF-modules. Composition mechanism supports reassembling of isolated code fragments in easy and useful way. Usage of SoC principles makes possible to decrease coupling in LSS, to decrease code redundancy, to reuse isolated CF-modules, to configure system by add/remove functionality if needed.

Finally, the existing POOTs provide SoC principles and offer a lot of toolkits to manage CF-problem in an effective way.

3 Post Object-Oriented Technologies: Main Features and Results of Comparative Analysis

As already mentioned above (see in Section 1) nowadays there are 3 main well-defined approaches in POOT-domain, namely: aspect-oriented software development (AOSD) [5], feature-oriented software development (FOSD) [6] and context-oriented software development technology (COSD) [7]. In order to reflect their essential features with respect to the problem of CF it is useful to represent an interaction between basic components of OOA and POOT [20].

AOSD was proposed in Research Center Xerox/PARC and it is now implemented in many programming languages such as Java / AspectJ, C ++, .NET, Python, JavaScript and some others [4]. AOSD allows to concentrate CF in separate modules called *aspects*, which should be localized in source code infected with CF using such means as points of *intersection* (point-cut) and *injection* (injection). Schematically this interaction is shown in Fig. 2, (a), where the white vertical rectangles C1, C2, C3 represent OOP-classes and gray horizontal rectangles A1, A2, A3 represent the aspects.

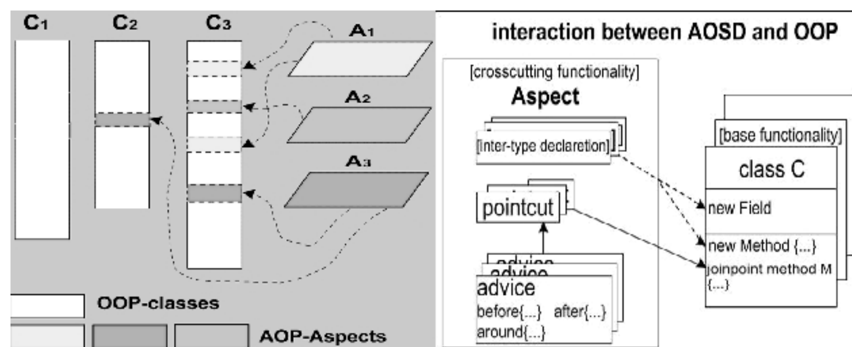

Fig. 2. AOSD: (a) – the conceptual scheme; (б) – the implementation facets (compare with [19])

More detailed the structure of aspect is shown in Fig. 2, (b). Any *aspect* consists of interconnected point-cut, of a *notification* (advice), and of an *introduction* (inner declaration). The task of *point-cut* is to define a connection point between *aspects* and basic methods in OOA-classes, in other words, *point-cut* determines those lines of code in the OOA-methods, were *notification* code has be introduced. A *notification* is a peace of code in OOA-language (e.g. in Java), which implements an appropriate CF, therefore notifications can be of three types: *before* – such a notification is perform before to call a OOA-method; *after* - a notification is made after this call; and *around* - a notification is executed instead to call a OOA-method. Also AOSD allows the introduction in OOA-classes new fields and methods that can be defined in aspects.

In the same way the FOSD and COSD schematically can be represented and analyzed carefully (see in [20] for more details). The results of this comparative analysis are presented in the Table 1.

Table 1. Results of comparative analysis for different POOT

POOT features / Estimation marks	Type of POOT		
	AOSD	FOSD	COSD
Modeling CF features at a higher level of abstraction	+	+	+
Implementation of homogeneous CF	+	+/–	+/–
Implementation of heterogeneous CF	+/–	+	+
Provide CF layers separately from a OOA-class	+	+	+
Context-dependent activation/deactivation of layers	–	–	+
Possibility to use several approaches simultaneously	+ /–	+ /–	–
Availability of CASE-tools to support this POOT	+	+	+ /–

Even a cursory analysis of this comparison shows that for a decision on the appropriateness and effectiveness of using an appropriate POOT to solve CF-problem in given LSS, it is necessary to take into account a number of other additional factors, which will be considered in the proposed approach.

4 Knowledge-Based Framework for Effectiveness's Estimation of Post Object-Oriented Technologies

Taking into account the results of performed analysis (see Section 2), and basing on some modern trends in the domain of POOT-development (see Section 3), we propose to elaborate a knowledge-based framework for comprehensive estimation of POOT-effectiveness to use them in software maintenance. Thus we proceed from one of possible definition of the term "knowledge" within the knowledge management domain [23], namely: *a knowledge is a collection of structured information objects and relationships combined with appropriate semantic rules for their processing in order to get new proven facts about a given problem domain.*

Then our next task is to define and to structure all information sources, and to elaborate appropriate algorithms and tools to process them with respect to the final

goal: how to estimate usage effectiveness of different POOTs in software maintenance.

4.1 Multi-dimensional model for POOT effectiveness's estimation

To implement the proposed knowledge-based approach the multi-dimensional modeling space is proposed in [20], and its graphical interpretation is shown in Fig. 3. According to this model the integrated effectiveness level is depend on two main interplaying factors, namely: 1) what type of LSS has to be modified with usage of an appropriate POOT; 2) what kind of POOT is used to eliminate the CF in this LSS. In order to answer these questions the following list of prioritized tasks can be composed:

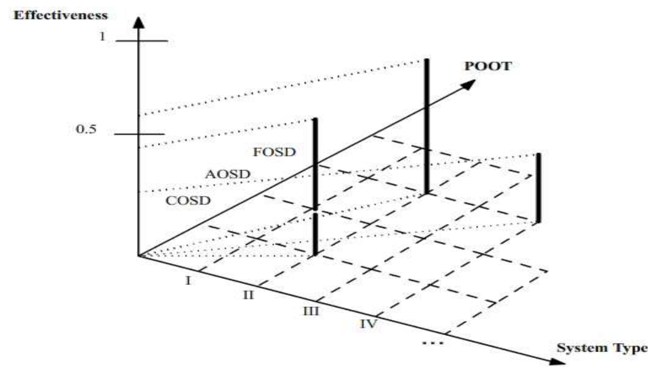

Fig. 3. 3-D modeling space for POOT's effectiveness estimation

- (i) to define a type of given LSS with respect to its structure complexity and to behavior of requirements, which this LSS in maintenance process is facing with;
- (ii) to calculate an average effort values for different POOT, if this one is used to eliminate CF in an appropriate LSS;
- (iii) to elaborate the metrics for CF assessment before and after LSS modification using a given POOT;
- (iv) to propose an approach to final effectiveness estimation of POOT's usage taking into account the results provided by activities (i) – (iii).

Below these tasks are solved sequentially, using knowledge-based and expert-centered methods and tools.

4.2 Definition of legacy software system types

To solve task (i) from the their list given in Section 4.1 the approach to analyzing and assessments of LSS's type proposed in [24] can be used, which is based on the following terms and definitions.

Def#1. *System Type* (ST) is an integrated characteristic of any LSS given as a tuple:

$$ST =< Structural\,Complexity\,, Requiremen\,tRank > \quad \textbf{(1)}$$

The first parameter estimates a complexity level of a given LSS, and the second one represents status of its requirements: their static features and dynamic behavior.

To calculate structural complexity (SC) the following collection of metrics was choused: *Cyclomatic Complexity* (V), *Weighted Method Complexity* (WMC), *Lack of Cohesion Methods* (LCOM), *Coupling Between Objects* (CBO), *Response For Class* (RFC), *Instability* (I), *Abstractness* (A), *Distance from main sequence* (D). The final value of SC can be calculated using formula (2), where the appropriate weighted coefficients for each metric were calculated in [24] with help of Analytic Hierarchy Process method [25].

$$\begin{aligned} SC \;=\; & K_V\,avgV \;+ K_{WMC}\,avgWMC \;+ K_{LCOM}\,avgLCOM \;+ K_{CBO}\,avgCBO \;+ \\ & + K_{RFC}\,avgRFC \;+ K_I\,avgI \;+ K_A\,avgA \;+ K_D\,avgD \end{aligned} \quad \textbf{(2)}$$

To evaluate the final value of SC of given LSS in terms of an appropriate linguistic variable (LV): *"Low", "Medium", "High",* the following scale was elaborated [24]:

$$\begin{aligned} & SC_{Min} \le Low < \frac{2*SC_{Min} + SC_{Max}}{3} \\ & \frac{2*SC_{Min} + SC_{Max}}{3} \le Medium \le \frac{SC_{Min} + 2*SC_{Max}}{3} \\ & \frac{SC_{Min} + 2*SC_{Max}}{3} < High \le SC_{Max} \end{aligned} \quad \textbf{(3)}$$

To define the second parameter given in formula (1), two relevant features of any requirement were considered [24], namely: a grade of it's *Priority* and a level of it's *Complexity*.

Def#2. *Requirements Rank* is a qualitative characteristic of LSS defined as a tuple:

$$Requiremen\;tRank \;=< Priority,\; Complexty \;> \quad \textbf{(4)}$$

In [24] is mentioned that in modern requirement management systems (RMS) like IBM Rational Requisite Pro, CalibreRM and some others, the *Priority* and *Complexity* of requirements are usually characterized by experts in informal way, e.g. using such terms as: *"Low", "Medium", "High".* The real example of such interface in RMS is presented in Fig. 4, with requirement's attributes "*Priority*" and "*Complexity*" (or "Difficulty" in terms of RMS-technology).

Taking into account the definition for linguistic variable (LV) given in [26], the appropriate term-sets for LVs *Priority* and *Complexity* respectively were defined in [24] as follows:

Requirements:	Priority	Difficulty	Stability
SR1: Parse Java Code	High	High	Medium
SR2: Elaborate Lexer for Java 5	High	Medium	Medium
SR3: Recognize all java lexical structures	High	High	Medium
SR4: Possibitity to parse single file	Medium	Low	Medium
SR5: Possibitity to parse whole package	Medium	Medium	Medium
SR6: Collect code statistics	Low	Medium	Medium
SR7: Recognize Java Grammatic	High	High	Medium
<Click here to create a requirement>	Medium	Medium	Medium

Fig. 4. The list of requirements completed in RMS Rational Requisite Pro

$$X : Priority \ ; \ T(Priority) = \{"neutral", "actual", "immidiate"\} \quad (5)$$

$$X : Complexity \ ; \ T(Complexity) = \{"low", "medium", "high"\} \quad (6)$$

Basing on definitions (1) – (6), the mapping procedure between 2 attribute spaces was elaborated in [24]. These attribute spaces are defined with apprpopiate LVs, namely: the space *"Requirements Rank"* with axes *"Priority"* and *" Complexity"*; the space "*System Type*" with axes "*Requirements Rank*" and "*Structural Complexity*". This mapping procedure in details is presented in [24], and the final result of this approach is shown on Fig. 5. It illustrates the main advantages of the proposed approach, namely: 1) we are able to estimate current state of system requirements w.r.t. their static and dynamic features; 2) basing on this estimation, we can define an appropriate type of investigated software system (e.g., some LSS in maintenance process), taking into account it's structural complexity and dynamic requirements behavior as well.

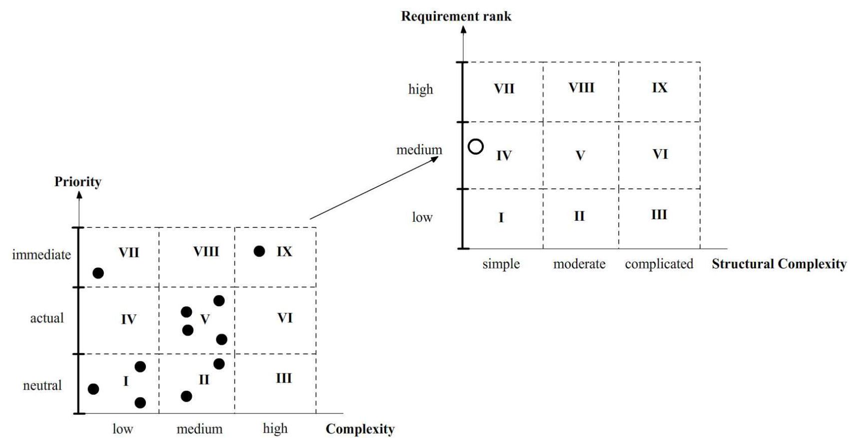

Fig. 5. (a) – the initial allocation of system's requirements in the space *"Requirement Rank"*; (b) – the mapped system's position in the space *"System Type"*

4.3 An architecture-centered method for POOT effort calculation

In order to solve task (ii) from their list given in Section 4.1 it is proposed to analyze basic architectural frames, which can be constructed for different POOT with usage of their OOP-specification. In [20] the following definition is proposed for this purpose.

Def#3. *Enhanced architectural primitive* (EAP) is a minimal-superfluous component-based scheme, which is needed to implement an interaction between basic OOP-elements (class, field, method) and specific functional POOT-elements.

Obviously, to perform the comparative analysis of different EAP in the correct way, they preliminary have to be represented in some uniform notation. As a such notation the architecture description language (ADL) should be used, because: 1) this notation is not depend on any specific programming tools; 2) in this way static and dynamic features of AP both can be described and analyzed.

The most important modeling abstracts of ADL (see e.g. in [27]) are *components*, *port*s and *connectors*, and there are such additional ADL - features as *role* and *interaction.* They have the following definitions within the context of this paper.

Def#4. *Component i*s a complex of functional items, which implements a certain part of a business logic in LSS, and which is supposed to have special interfaces (ports) for communication with other entities in an operating environment.

Def#5. *Port* is an interface to provide an interaction between several components.

Def#6. *Connector* is a special architectural item to join ports of different components.

Def#7. *Role* is a special feature of a given connector to identify its communicating ports.

Def#8. *Interaction* is a special feature of given connector defined using its roles.

More detailed the notion *port* can be characterized in the following way: 1) there is so-called *single port* - this is an interface of any component to communicate with some another one via exactly one connector; 2) furthermore there is a *case-port* - this is an interface of any component to communicate with another components via more then one connectors (e.g., using an appropriate Boolean variable as a flag to switch communication, etc.). Similarly, the notion *connector* can be classified as follows: 1) a *binary connector* – this is a connector with 2 fixed roles only; 2) a *multiply connector* – this is a connector, which has exactly 1 input role and more then 1 output roles; 3) a *case connector* – this kind of connectors can have a lot of input and output role as well.

Using the definitions Def#3 – Def#8 the appropriate EAP for all mentioned above POOT were elaborated [20]. As one example the EAP for AOP is shown on Fig. 6, which reflects how the specific AOP-features such as *advice* and *inner declaration* (they are shown as rectangular icons in grey color) are interacting with basic OOP – elements, namely: *class, field* and *method* (they are represented as crosswise icons in white color).

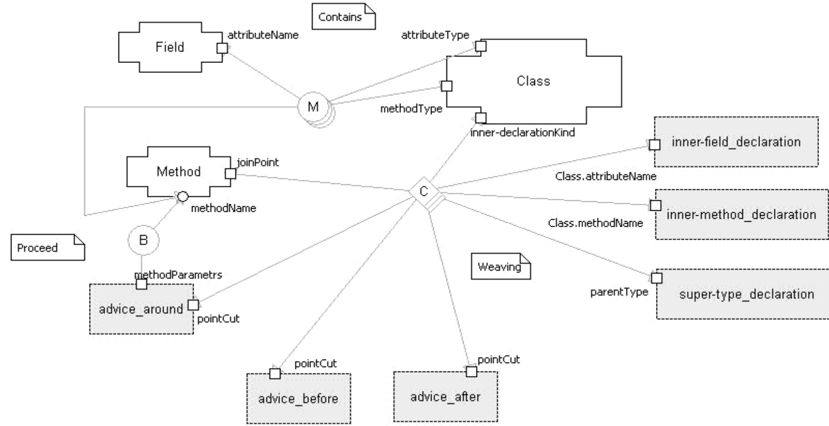

Fig. 6. ADL-specification for the aspect-oriented EAP

To calculate the complexity coefficients (CC) of the elaborated EAP the following formulas are proposed in [20], namely:

$$Component = 0.6 * \# POOT + 0.4 * \# OOP, \tag{7}$$

where a $Component$ is the CC of an appropriate EAP, $\# OOP$ is a number of architectural OOP – components, and $\# POOT$ is a number of POOT – components included in this EAP. These values are multiplied with the weight coefficients: *0,6* and *0,4* respectively, and these coefficients can be defined using some expert methods (see in [20] for more details);

$$Connector = 0.2 * \# BinaryConnector + 0.3 * \# MultyConnector + 0.5 * \# CaseConnector, \tag{7}$$

where a $Connector$ is the CC of connectors included in an appropriate EAP, which is calculated using the number of binary connectors: $\# BinaryConnector$, the number of multi-connectors $\# MultyConnector$ and the number of case-connectors: $\# CaseConnector$, with respect to the appropriate weight coefficients *0.2, 0.3* and *0.5,* which also are defined by some experts [20];

$$Port = 0.8 * \# SinglePort + 0.7 * \# CasePort, \tag{8}$$

where $Port$ is the CC of ports included in an appropriate EAP, which takes into account the number of single ports: $\# SinglePort$, and the number of case ports: $\# CasePort$ with appropriate weigh coefficients.

Using formulas (7) – (9) the summarized value $Complexity$ of an appropriate EAP, measured in so-called architectural units (a.u.) [20] can be calculated as follows:

$$Complexity = Component + Connector + Port \quad \textbf{(9)}$$

The final values of CC for all POOT were calculated using formula (10), and they are represented in Table 2 (see in [20] for more details).

Table 2. The values of architectural complexity for the different POOT

POOT type	CC for components (a.u.)	CC for connectors (a.u.)	CC for ports (a.u.)	Summarized values of CC (a.u.)
AOSD	4,8	1	4,3	10,1
FOSD	3,6	1	3,9	8,5
COSD	2,8	0,7	4,1	7,6

Basing on the estimation values aggregated in Table 2 it is possible to make conclusions about average implementation efforts by usage of appropriate POOT to solve CF-problems in legacy software systems within their maintenance.

4.4 Quantitative metrics for crosscutting in legacy software

There are different ways to characterize a nature of the CF and it's impact to software source code. A number of studies are dedicated to a classification, qualitative and quantitative description of CF problem [3,14-16]. The aim of our research is to assess an impact, which CF makes to a structure of OOP-based software system during it's evolution in maintenance; therefore we are focusing on quantitative facet of crosscutting nature. To reach this goal it is proposed to perform next three steps.

Step 1: Localize source code belonged to a particular CF in a given LSS. Although exists several source code analysis tools for CF localization, e.g., tool CIDE [28], this problem remains really complicated for autoimmunization and demands an expert in code structure and business-logic of an appropriate LSS.

Step 2: Calculate a specific crosscutting weight ratio of a particular CF in the system indicated as CF_{ratio} [20]. This coefficient shows a ratio between OOP-classes, "damaged" by a particular CF and all OOP-classes in the system, or it's projection, e.g. business logic realization without subordinate classes of a framework. This coefficient possible to represent as

$$CF_{ratio} = \frac{C_{cf}}{C_{cf} + C} \ , \quad \textbf{(10)}$$

where C_{cf} – number of classes in LSS, "damaged" with CF, C – number of classes free of CF. Obviously, that $CF_{ratio} \in [0;1]$, and if $CF_{ratio} = 0$, it means a particular

functionality is not crosscutting; and if CF_{ratio} = 1, it means all classes are "damaged" with a particular CF.

Step 3: Calculate a residual crosscutting ratio indicated as $\mathrm{RCR}_{\mathrm{ratio}}$. This metric, based on DOS (Degree of Scattering) value, proposed in [14], namely "…*DOS is normalized to be between 0 (completely localized) and 1 (completely delocalized, uniformly distributed)*". Nevertheless this metric does not allow to asses "damage" degree, done by a particular CF, therefore we propose to refine DOS-metric in following way

$$RCR_{ratio} = DOS \cdot CF_{ratio}, \quad \textbf{(11)}$$

where DOS – Degree of Scattering; CF_{ratio} – specific crosscutting weight ratio of a particular CF. Similarly to CF_{ratio}, $RCR_{ratio} \in [0;1]$, if RCR_{ratio} = 0, it means that CF is localized in a separate module and it is no more crosscutting; if RCR_{ratio} = 1, it means that CF effects a whole system and is uniformly distributed.

Thus the proposed quantitative metrics (11) – (12) give to an expert a possibility to assess a distribution nature of a CF, and to estimate a "CF-damage" for a given LSS.

4.5 Fuzzy logic approach to complex effectiveness estimation of POOT

Based on assessment of POOT average implementation efforts (see Chapter 4.3), and assessment for residual crosscutting ratio (see Chapter 4.4) it is possible to estimate an integrated effectiveness of POOT usage. Although because of different scale and units of measurement for proposed assessments, it is hard to evaluate them within a single analytical method. Therefore, for further evaluations it is proposed to use one of algorithms of the fuzzy logic [26], namely the Mamdani's algorithm, which consists of 6 steps. According to this algorithm to estimate effectiveness of POOT usage it is necessary to compose fuzzy production rules (FPR). In this paper a verbal description for these rules is omitted, instead of this the widespread symbolic identifiers for short description of FPR are listed in Table 3.

Table 3. A symbolic representation form for the description for FPR

Symbolic form	Description
Z	Zero
PS	Positive Small
PM	Positive Middle
PB	Positive Big
PH	Positive Huge

The whole system of elaborated FPR consists of 20 definitions (see in [29] for more details), and the fragment of this FPR-system is listed below:

1. RULE_1: If "β_1 is **PS**" and "β_2 is **Z**", then "β_3 is **Z**";

2. RULE_2: If “ β_1 is **PM**” and “ β_2 is **Z**”, then “ β_3 is **Z**”;
3. …
4. RULE_9: If “ β_1 is **PS**” and “ β_2 is **PM**”, then “ β_3 is **PM**”;
5. …

Corresponding to the Mamdani’s algorithm, the next step is a fuzzifying of variables in FPR, therefore average implementation efforts, residual crosscutting ratio, and effectiveness of POOT usage have to be represented as LV. The output LV E_{POOT} is the effectiveness of POOT-usage, the LV E_{POOT} is bounded on universe X, and it belongs to the interval [0;1]. The term set for this LV looks like:

$E_{POOT} \in \{non-effective, low-effective, mid-effective, effective, very-effective\}$,

and it could be represented in short form as $E_{POOT} \in \{Z, PS, PM, PB, PH\}$. The corresponding identifier for E_{POOT} is β_3 (see FPR above), and it is shown in Fig. 7.

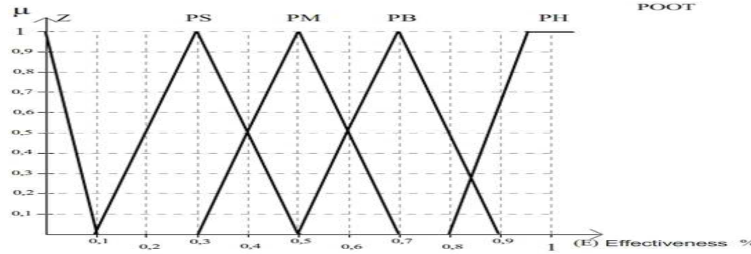

Fig. 7. The graphic form for LV “Effectiveness” E_{POOT}

The input LV C_{POOT} represents average implementation efforts, C_{POOT} is bounded on universe X and belongs to an interval [(EAP)min; (EAP)max], where EAPmin, EAPmax are minimum and maximum values of architectural complexity (measured in a.u.) for appropriate LSS type respectively. The term set for the C_{POOT} linguistic variable (LV) looks like: $C_{POOT} \in \{low, middle, high, huge\}$ and could be represented in short form $C_{POOT} \in \{PS, PM, PB, PH\}$. The corresponding identifier for C_{POOT} is β_1 (see FPR above). The graphical interpretation for this LV is similar to the graphic, depicted on Fig. 7.

The input LV P_{POOT} is a residual crosscutting ratio (see formula (12)). The LV P_{POOT} is bounded on universe X and belongs to interval [0;1]. The term set for this variable looks like: $P_{POOT} \in \{useless, low, middle, high, huge\}$, and it could be represented in short form as $P_{POOT} \in \{Z, PS, PM, PB, PH\}$. The corresponding identifier for P_{POOT} is β_2 (see FPR-system above). The visual interpretation is similar to the graphic depicted in Fig. 7.

5 The Test-Case and Result Discussion for the Proposed Approach

To illustrate the proposed approach the real LSS for personal data management was analyzed [29]. It consists of 15 java-classes, and it contains a homogenous realization of "logging" crosscutting functionality. Accordingly to the LSS – type definition method (see Section 4.2) this application belongs to the III-rd system type with rank: {"Low structural complexity"; "High requirement rank"}. The source code of this LSS was sequentially modified using 3 POOT: AOSD, FOSD, and COSD respectively. The final results of POOT effectiveness estimation are shown in Table 4. The first column lists all LSS – modifications to be compared: an initial OOP - version, which has to be re-structured wit respect to CF-problem, and its 3 modifications done with usage of different POOT. In the second column the summarized efforts needed for these modifications with respect to architectural-centered complexity are calculated (see Section 4.3). The data given in the third column of Table 4 show the level residual crosscutting ratio which is presented (for initial OOP-version) or which is remained after its redesigning with the appropriate POOT. The forth column indicates the final effectiveness's estimation values for all LSS-versions.

Table 4. Effectiveness of usage of POOT in a target system

(P)OOT	Architectural complexity (a.u.)	Residual crosscutting ratio (%)	Effectiveness level (%)
OOP	122.51	69.52	6,7
AOSD	79.43	0,15	73,3
FOSD	116.16	29.06	34,4
COSD	115.88	8.78	32,8

The results achieved show, that OOP actually is not enough effective to solve crosscutting problem (done with 6.7% only). The most preferable approach to eliminate this issue in the given type of LSS (as mentioned above, this is the III-rd system type according to LSS-classification proposed in Section 4.2), is an AOSD which provides effectiveness level over than 70%.

It is also to mention, although an effectiveness level of COSD and FOSD is lower than AOSD, over 30% for homogenous CF, it is still much better result than OOP. Taking into account a qualitative advantage of these two another technologies, namely: a possibility to implement a heterogeneous CF also (see Table 1), it can be reasonable to use one of them for LSS-maintenance to deal with such kind of CF in much effective way than AOSD.

6 Conclusions and Future Work

In this paper we have presented the intelligent approach to effectiveness's estimation of modern post object-oriented technologies (POOT) in software development, which

aims to utilize domain-specific knowledge for this purpose. This knowledge base includes such important and interconnected data resources as: 1) structural complexity of legacy software; 2) dynamic behavior of user's requirements; 3) architectural-centered implementation efforts of different POOT. To process these data the quantitative metrics and expert-oriented estimation algorithms were elaborated. The final complex estimation values of POOT's effectiveness assessment are defined using fuzzy logic method, which was successfully tested on some real-life legacy software applications.

In future we are going to extend a collection of metrics for POOT-features assessment, and to apply some alternative (to fuzzy logic method) approaches to final decision making. Besides that it is supposed to develop an appropriate software CASE-tool for expert's data handling in the proposed knowledge-based estimation framework.

7 References

1. Sommerville, I.: Software Engineering. Addison Wesley (2011)
2. Eilam, E.: Reversing: Secrets of Reverse Engineering. Wiley Publishing (2005)
3. Sven Apel et al. On the Structure of Crosscutting Concerns: Using Aspects of Collaboration? In: Workshop on Aspect-Oriented Product Line Engineering (2006)
4. Przybyłek, A.: Post Object-oriented Paradigms in Software Development: A Comparative Analysis. In: Proceedings of the International Multi-conference on Computer Science and Information Technology, pp. 1009-1020 (2007)
5. Official Web-site of Aspect-oriented Software Development community, http://aosd.net
6. Official Web-site of Feature-oriented Software Development community, http://fosd.de
7. Official Web-site of Context-oriented Software Development group, http://www.hpi.uni-potsdam.de/hirschfeld/cop/events
8. Highsmith, J.: Agile Project Management. Addison-Wesley (2004)
9. Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (2001)
10. Sheldon, T., Jerath, Kh., Chung, H.: Metrics for Maintainability of Class Inheritance Hierarchies. J. of Software Maintenance and Evolution, Vol. 14, pp. 1--14 (2002)
11. Harrison, R. Counsell, S.J.: The Role of Inheritance in the Maintainability of Object-Oriented Systems. In: Proceedings of ESCOM '98, pp. 449--457 (1998)
12. Aversano, L. Cerulo, L. Penta, M. Di.: The Relationship between Design Patterns Defects and Crosscutting Concern Scattering Degree: An Empirical Study. J. IET Software, vol. 3, pp. 395--409 (2009)
13. Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In: Proceedings of OOPSLA'02, pp. 161--173 (2002)
14. Eaddy, M. et al.: Do Crosscutting Concerns Cause Defects? In: IEEE Trans. Softw. Eng., 34(4), pp. 497--515 (2008)
15. Filman, R., Elrad, S. Aksit. M.: Aspect-Oriented Software Development. Addison Wesley Professional (2004)
16. Figueiredo, E.: Concern-Oriented Heuristic Assessment of Design Stability. PhD thesis, Lancaster University (2009)
17. Official Web-site of MSDN, https://msdn.microsoft.com/en-us/library/ee658105.aspx
18. Clarket, S., et al.: Separating Concerns throughout the Development Lifecycle. In: Intl. Workshop on Aspect-Oriented Programming ECOOP (1999)

19. Apel, S.: The Role of Features and Aspects in Software Development. PhD thesis, Otto-von-Guericke University Magdeburg (2007)
20. Tkachuk, M., Nagornyi, K.: Towards Effectiveness Estimation of Post Object-oriented Technologies in Software Maintenance. J. Problems in Programming, vol. 2-3 (special issue), pp.252--260 (2010)
21. Taromirad M., Paige, M.: Agile Requirements Traceability Using Domain-Specific Modeling Languages. In: Extreme Modeling Workshop, pp. 45--50 (2012)
22. Tarr, P.L., et al.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proceedings of the International Conference on Software Engineering (ICSE), ACM, Los Angeles, USA, pp. 107--119 (1999)
23. Official Web-site of System Thinking World community, http://www.systems-thinking.org/kmgmt/kmgmt.htm
24. Tkachuk M., Martinkus I.: Models and Tools for Multi-dimensional Approach to Requirements Behavior Analysis. In: H.C. Mayr et al. (eds.) UNISCON 2012, LNBIP vol. 137, pp. 191--198. Springer-Verlag, Heidelberg (2013)
25. Saaty, T.L.: Fundamentals of the Analytic Hierarchy Process. RWS Publications (2000)
26. Zadeh, L.A.: Fuzzy Sets. WorldSciBook (1976)
27. Garlan, D., Monroe, R., Wile, D.: ACME: An Architecture Description Interchange Language. In: Proceedings of CASCON'97, p.p. 169--183, Toronto, Canada (1997)
28. Official Web-site of CIDE-project, http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/
29. Nagornyi, K.: Elaboration and Usage of Method for Post Object-oriented Technologies Effectiveness's Assessment. J. East-European on Advanced Technologies, vol. 63, p.p. 21--25 (in Russian) (2013)