

Models, Methods and Tools for Effectiveness Estimation of Post Object-Oriented Technologies in Software Maintenance

Mykola Tkachuk^(✉), Konstantyn Nagorny, and Rustam Gamzayev

National Technical University “Kharkiv Polytechnic Institute”, Frunze Str., 21,
Kharkiv 61002, Ukraine
tka@kpi.kharkov.ua, k.nagorny@gmail.com,
rustam.gamzayev@gmail.com

Abstract. An intelligent framework for effectiveness 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 requirements; (3) architecture-centered implementation issues by usage of different POOT. These 3 components are formalized and combined in form of the algorithmic model, and the final estimation values of POOT effectiveness are defined using fuzzy logic method and CASE-tools elaborated, which were tested successfully at the maintenance case-study of real-life software application.

Keywords: Post object-oriented technology · Effectiveness · Crosscutting functionality · Knowledge-based approach · Algorithmic model · Metrics · Fuzzy logic · CASE-tool

1 Introduction: Problem Domain Actuality and Research Aims

Nowadays the most part of real-life software systems are designed and implemented using the object-oriented programming (OOP) paradigm [1]. A well-known and important problem of support and maintenance for such applications is often modifications of many of their subsystems and a need to develop new components for additional business logic, taking into account new user requirements. In order to emphasize this issue we propose to use in this paper the notion of “legacy software system” (LSS), similarly to 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 last ten years some post object-oriented technologies (POOT) were elaborated and became intensively used in development process, 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 at the same time they have additional features, which allow solving the crosscutting problem electively. From the other hand usage of any of POOT for LSS maintenance and reengineering is concerned with an additional time and other efforts in software development. That is why many researchers emphasize an actual need to elaborate appropriate approaches for complex estimation of POOT 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 are also can be characterized as “post object- oriented” approaches.

Taking into account issues mentioned above, the main objective of the research presented in this paper is to propose an intelligent complex approach to effectiveness estimation of POOTs usage in software maintenance. The rest of this paper is organized in the following way: Sect. 2 analyses some critical issues in OOP and reflects the phenomena of crosscutting functionality in software maintenance. In Sect. 3 existent POOTs are analyzed and results of their comparing are shown with respect to software maintenance problems. In Sect. 4 we present the knowledge-based approach for effectiveness 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 results of test-case for the proposed approach. In Sect. 6 the paper concludes with a short summary and with an outlook on next steps to be done in the proposed R&D approach.

2 Crosscutting Functionality Phenomena in Software Maintenance: Related Work

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 of a source code and doing software system re-design, developers face with “bottlenecks” of OOP: increase coupling among classes [10]; increase of depth inheritance tree (DIT) for classes hierarchies [11]; modification of design patterns instances [12, 13]; emerging lack of modularity in functionality realization [14].

A number of studies investigate problems of OOP mentioned above, and their 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 phenomenon. Crosscutting concern (hereby referred as “cross-cutting functionality” - CF) is a concern emerged on user requirements level and often crosscuts on software design level, this is a part of a business logic, which can not be localized in the separate module on source code view, but it stays separate on requirement view [15]. In literature exist 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 a system source code (see e.g. in [13–16]). There are some widespread examples of software system features which could be considered 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 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).

<pre> 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 </pre>	<pre> 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 </pre>
--	---

Fig. 1. Crosscutting functionality types

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

- CF complicates traceability of various software design artifacts, e.g. requirement 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.

These common negative CF-features cause specific problems in maintenance of a given LSS, which also are considered in some already existent publications. E.g., according to the research presented in [22] there is a strong positive correlation (with Spearman's rank-order coefficient approx. from 0.650 to 0.744) between the degree of scattering and the number of defects in LSS source code. Authors [23] also report about the correlation between the number of defects in design-pattern classes of LSS and scattering of induced crosscutting concerns. Another study on CF-consequences with respect to the quality of maintenance process deals with modularity problem in source code [24]. From the other hand, exactly because of these problems arose, attempts to improve CF-issues in maintenance exist, e.g. a model-based approach to reuse cross-cutting concerns [25], and some others.

A conceptual approach, which allows dealing with CF, is the separation of concerns (SoC) [26]. 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 make possible to decrease coupling in LSS, to decrease code redundancy, to reuse isolated CF-modules, to configure system by add/remove functionality if it is needed.

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

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

As already mentioned above (see in Sect. 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 now it is implemented in many programming languages such as Java/AspectJ, C++, .NET, Python, JavaScript and some others [4]. AOSD allows concentrating 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 aspect-modules.

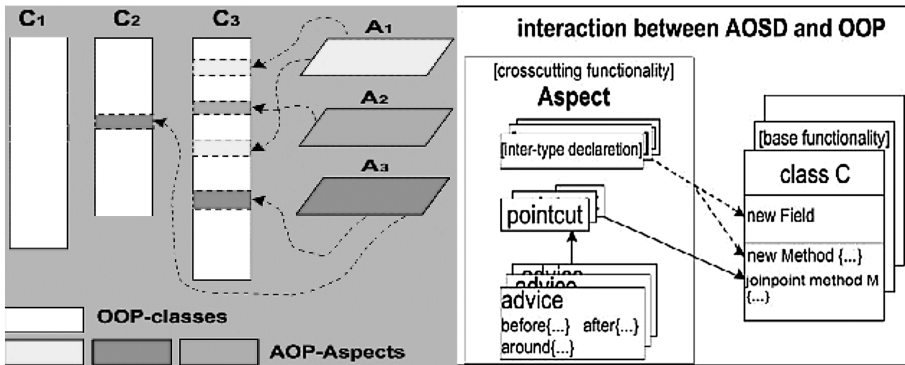


Fig. 2. AOSD: (a) the conceptual scheme; (b) the implementation facets (to compare with [19])

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, where *notification* code has to be introduced. A *notification* is a piece of code in OOA-language (e.g. in Java), which implements an appropriate CF, therefore notifications can belong to three types: *before* – a notification is performed before an OOA-method is called; *after* – a notification acts after this call; and *around* – a notification is executed instead of an OOA-method calling. Also AOSD allows the introduction into 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 result of this comparative analysis is presented in the Table 1.

Table 1. Results of comparative analysis for several 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 making concerning 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 Approach to Effectiveness's Estimation of Post Object-Oriented Technologies

Taking into account results of performed analysis (see Sect. 2), and basing on some modern trends in the domain of POOT-development (see Sect. 3) we propose to elaborate a knowledge-based approach 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 [27], 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 Metaphor of Multi-dimensional Information Space and Algorithmic Model for POOT Effectiveness Estimation

To elaborate the proposed knowledge-based approach a metaphor of the multi-dimensional modeling space is proposed in [20], and its simplified graphical interpretation is shown in Fig. 3.

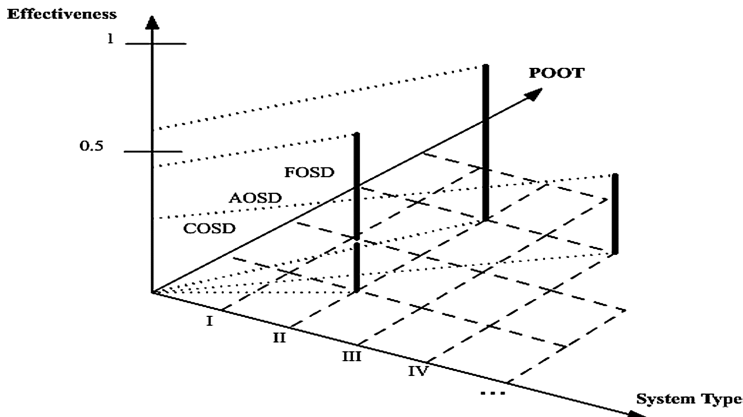


Fig. 3. Metaphor of the multi-dimension space for POOT effectiveness estimation

According to this model the integrated POOT effectiveness level depends of 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 actually used to eliminate the CF in this LSS. In order to answer these questions the following list of prioritized tasks can be composed:

- (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 average effort values for different POOTs, if they were used to eliminate CF in an appropriate LSS;
- (iii) to elaborate metrics for CF assessment before and after LSS modification using a given POOT;
- (iv) to propose an approach for final effectiveness estimation of POOT usage taking into account results, provided by activities (i) – (iii).

In order to generalize the proposed knowledge-based framework described as steps (i)–(iv) it is useful to utilize the algorithmic modeling approach (e.g. see in [28]). This one can be given as the following tuple:

$$AM = \langle InfoSpace, Workflow(Methods), CFMetrics \rangle \quad (1)$$

where the *InfoSpace* is the multi-dimensional informational space (the part of this one is shown in Fig. 3), the *Workflow(Methods)* are algorithms which implement appropriate methods to define all values requested for the final POOT effectiveness estimation, and *CFMetrics* is the collection of specific metrics for the CF-assessment. Basing on the formula (1) it is possible to evaluate several aspects of CF-issues in LSS by usage of complex and heterogeneous data structures, to process them with various algorithms, and to estimate achieved results with appropriate quantitative or qualitative metrics.

Tasks (i) – (iv) are solved sequentially below, using knowledge-based and expert-centered methods and relevant software tools.

4.2 Definition of Legacy Software System Types

To solve task (i) from the list given in Sect. 4.1 the approach for analyzing and assessment of LSS's type which is proposed in [29] can be used, which is based on following terms and definitions.

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

$$ST = \langle StructuralComplexity, RequirementRank \rangle \quad (2)$$

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 chosen: *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 appropriate weighted coefficients for each metric were calculated in [29] with a help of the Analytic Hierarchy Process method [30].

$$SC = K_V avg V + K_{WMC} avg WMC + K_{LCOM} avg LCOM + K_{CBO} avg CBO + K_{RFC} avg RFC + K_I avg I + K_A avg A + K_D avg D \quad (3)$$

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 [29]:

$$\begin{aligned} SC_{Min} \leq Low &< \frac{2 \cdot SC_{Min} + SC_{Max}}{3} \\ \frac{2 \cdot SC_{Min} + SC_{Max}}{3} &\leq Medium \leq \frac{SC_{Min} + 2 \cdot SC_{Max}}{3} \\ \frac{SC_{Min} + 2 \cdot SC_{Max}}{3} &< High \leq SC_{Max} \end{aligned} \quad (4)$$

To define the second parameter given in formula (1), two relevant features of any requirement were considered [29], namely: a grade of its *Priority* and a level of its *Complexity*.

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

$$RequirementRank = \langle Priority, Complexity \rangle \quad (5)$$

It is to note that in modern requirement management systems (RMS) like IBM Rational Requisite Pro, CalibreRM and some others, the *Priority* and the *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).

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: Possibility to parse single file	Medium	Low	Medium
SR5: Possibility to parse whole package	Medium	Medium	Medium
SR6: Collect code statistics	Low	Medium	Medium
SR7: Recognize Java Grammar	High	High	Medium
* <Click here to create a requirement>	Medium	Medium	Medium

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

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 [29] as follows:

$$X:Priority; T(Priority) = \{“neutral”, “actual”, “immediate”\} \quad (6)$$

$$X:Complexity; T(Complexity) = \{“low”, “medium”, “high”\} \quad (7)$$

Basing on definitions (2) – (7), the mapping procedure between 2 attribute spaces was elaborated in [29]. These attribute spaces are defined with appropriate 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 [29], 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 its structural complexity and dynamic requirements behavior as well.

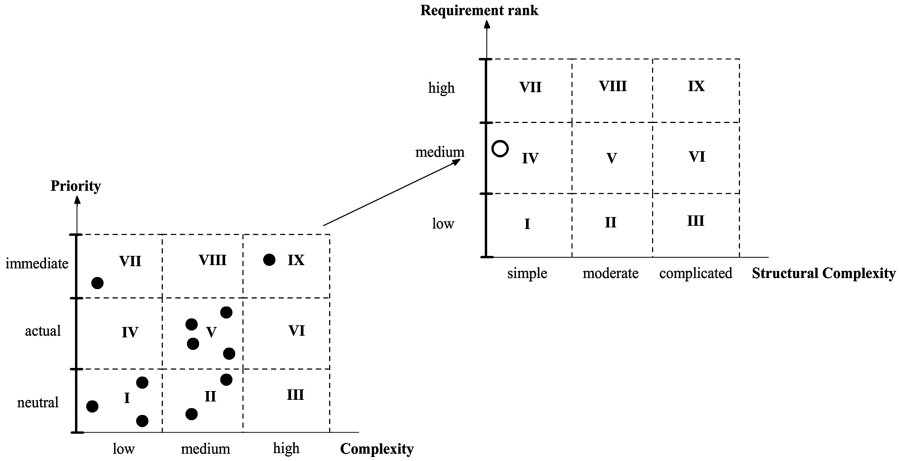


Fig. 5. (a) initial allocation of system requirements in the space “*Requirement Rank*”; (b) mapped system position in the space “*System Type*”

Further, according to formula (1), we have to elaborate next methods and metrics, listed in the following sections.

4.3 An Architecture-Centered Method for POOT Effort Calculation

In order to solve task (ii) from their list given in Sect. 4.1 it is proposed to analyze basic architectural frames, which can be constructed for different POOTs 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, preliminary they have to be represented in some uniform notation. As such notation the architecture description language (ADL) family should be used, because: (1) this notation does not depend on any specific programming tools; (2) in this way static and dynamic features of AP both can be described and analyzed.

It is proposed to use Acme-ADL, because it operates with an irredundant set of basic abstracts, for static modeling (see e.g. in [31]), such as: *components*, *ports* and

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

$$Component = 0.6 \cdot \#POOT + 0.4 \cdot \#OOP \quad (8)$$

where the *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 weight coefficients: 0,6 and 0,4 respectively, and these coefficients can be defined using some expert methods (see in [20] for more details).

$$Connector = 0.2 \cdot \#BinCon + 0.3 \cdot \#MultyCon + 0.5 \cdot \#CaseCon \quad (9)$$

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

$$Port = 0.8 \cdot \#SinglePort + 0.7 \cdot \#CasePort \quad (10)$$

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

Using formulas (8)–(10) a summarized value the *Complexity* of an appropriate EAP, measured in so-called architectural units (a.u.) [20] can be calculated as follows:

$$Complexity = Component + Connector + Port \quad (11)$$

The final value of CC for all POOTs, were calculated using the formula (11), and they are represented in Table 2 (see in [20] for more details).

Table 2. Values of an architectural complexity for 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

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

4.4 Quantitative Metrics for Crosscutting Functionality Ratio in Legacy Software

There are various ways to characterize a nature of the CF and its impact to software source code. A number of studies are dedicated to a classification, qualitative and quantitative

description of CF problems [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 its evolution in maintenance; therefore we are focusing on quantitative facet of a 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 there are exist several source code analysis tools for CF localization, e.g., tool CIDE [32], 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 its projection, e.g. business-logic realization without subordinate classes of a framework. This coefficient possible to represent as

$$CF_{ratio} = \frac{C_{cf}}{C_{cf} + C} \quad (12)$$

where C_{cf} – a number of classes in LSS, “damaged” with CF, C – a number of classes free of CF. Obviously $CF_{ratio} \in [0;1]$, and if $CF_{ratio} = 0$, it means that a particular functionality is not crosscutting; and if $CF_{ratio} = 1$, it means that all classes are “damaged” with a particular CF.

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

$$RCR_{ratio} = DOS \cdot CF_{ratio} \quad (13)$$

where DOS – Degree of Scattering; CF_{ratio} – the 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 affects the whole system and it is uniformly distributed.

Thus proposed quantitative metrics (12) – (13) provide to experts a possibility to assess a distribution nature of a CF, and to estimate a “CF-damage” in a given LSS.

4.5 Fuzzy Logic Approach to Complex Effectiveness Estimation of POOT

Based on assessment of a POOT average implementation efforts (see Sect. 4.3), and assessment for the residual CF ratio (see Sect. 4.4) it is possible to estimate an integrated effectiveness of a 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 [33], namely the Mamdani’s algorithm, which consists of 6 steps. According

to this algorithm 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. The symbolic representation form for the FPR description

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 [34] 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. RULE_9: If “ β_1 is **PS**” and “ β_2 is **PM**”, then “ β_3 is **PM**”;
4.

Corresponding to the Mamdani’s algorithm, the next step is fuzzifying of variables in FPR, therefore average implementation efforts, residual crosscutting ratio, and effectiveness of POOT usage have to be represented as a 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 a 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.

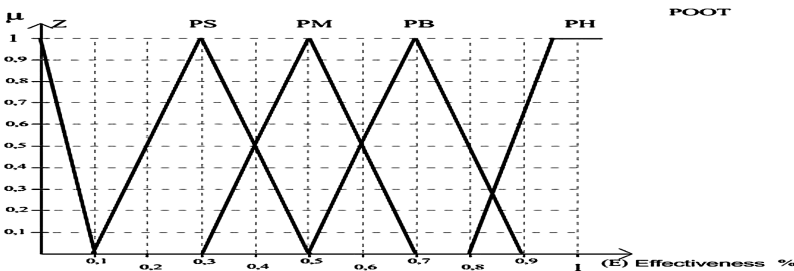


Fig. 7. The graphic form for LV “Effectiveness” E_{POOT}

The input LV C_{POOT} represents average implementation efforts, C_{POOT} is bounded to 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 an 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 a 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 the residual crosscutting ratio, see formula (13). The LV P_{POOT} is bounded to 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 a 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 First Implementation Issues, Test-Case and Results Discussion for the Proposed Approach

To prove the proposed approach an appropriate software tool was implemented, its functionality allows to automate the most routine data processing operations of the CF/LSS – estimation methods introduced in the previous Sect. 4. This tool is designated below as a CASE, and its interaction with such additional facilities as RMS and Java source code parser can be considered as the completed information technology (IT) to estimate POOT effectiveness.

5.1 An Informational Technology to Support the Proposed Approach

The proposed IT can be represented as the diagram in IDEF0 [34] notation, and it is depicted in Fig. 8. It consists of 6 functional blocks: 1. “*Requirements Rank assessment*”, 2. “*Structural Complexity calculation*”, 3. “*System Type definition and LSS modification with POOT*”, 4. “*POOT-modification Costs calculation*”, 5. “*Residual CF ratio calculation*”, 6. “*Estimation of POOT usage effectiveness*”. Each block, corresponding to the IDEF0 notation, has 4 kinds of interfaces: *Inputs* – arrows enter into a block from the left, they represent the data to be processed in this block; *Controls* – arrows enter from the top, they define all business logic algorithms and models; *Mechanisms* – arrows enter from below, and they reflect all implementation issues related to this block (including human actors, if needed); *Outputs* – arrows leave the block from the right, and they are the results of data processing in this block.

Block 1 operates with functional requirements, obtained from an RMS-system and the result of block 1 is a value of the *Requirement Rank*, represented as a linguistic variable. Block 2 analyzes Java source code of a given LSS and defines its value of *Structural Complexity*, which is also represented as a linguistic variable. Block 3 operates with both parameters: *Requirement Rank* and *Structural Complexity*, and allows to define a value of *System Type*. After this operation it is possible to perform

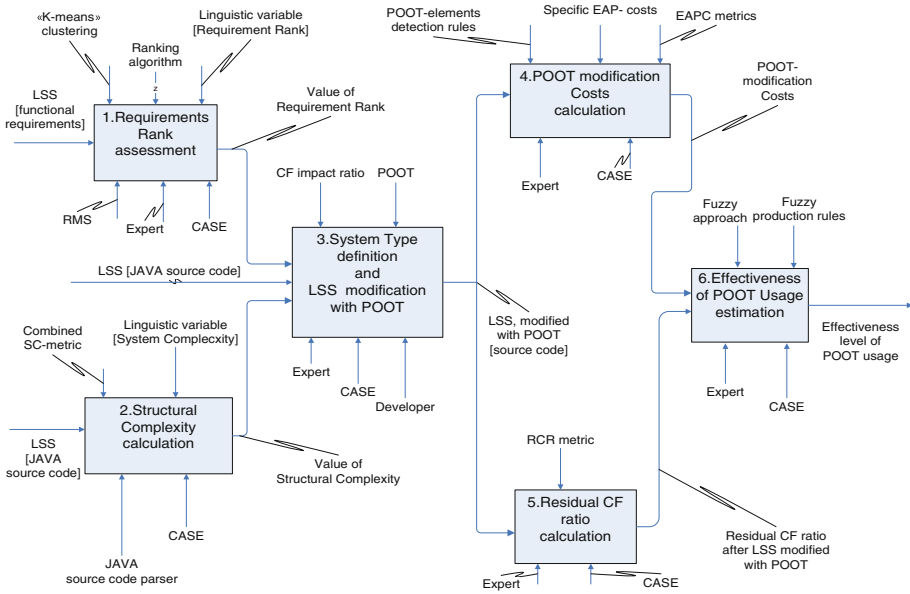


Fig. 8. The information technology scheme of a proposed approach

POOT-modification on LSS source code level, in order to localize and to isolate CF-concerns in separate POOT-modules. Functional blocks 4 and 5 deal with received POOT-modified source code to calculate values of *POOT modification costs* (block 4) and *Residual CF ratio* (block 5). Finally, block 6 realizes a fuzzy logic approach to assess the final value of POOT usage effectiveness. As the human actors the *Expert* (a person, who is interesting in POOT estimation results) and the *Developer* (a person, who is responsible for LSS maintenance) are supposed to participate in the data processing within the proposed IT-scheme (see Fig. 8).

5.2 Test-Case Data Description and Results Discussion

To illustrate the proposed approach the real LSS for personal data management was analyzed [35]. It consists of 115 java-classes, and it contains a homogenous realization of “logging” crosscutting functionality. Accordingly to the LSS – type definition method (see Sect. 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 several POOTs: AOSD, FOSD, and COSD respectively. Final results of the 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 with respect to the CF-problem, and its 3 modifications done with usage of a different POOT. In the second column there are summarized efforts needed for these modifications with respect to architectural-centered complexity are calculated (see Sect. 4.3). The data given in the third column of Table 4 show the level of the residual CF ratio which is presented (for initial OOP-version) or which is remained after its

Table 4. Effectiveness values of POOT usage in the target system

OOP / POOT	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

re-design with an appropriate POOT. The forth column indicates the final effectiveness estimation values for all LSS-versions.

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

It is also to mention, although the effectiveness level of COSD and FOSD is lower than AOSD, over 30 % for a 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 the LSS-maintenance to deal with such kind of CF in more effective way than AOSD.

6 Conclusions and Future Work

In this paper we have presented the intelligent approach to effectiveness estimation of modern post object-oriented technologies (POOT) in the software maintenance, which aims to utilize domain-specific knowledge for this purpose. This knowledge is complex and interconnected data resources organized in a form of the multi-dimensional information space, where the following characteristics can be defined: (1) the structural complexity of a legacy software; (2) the dynamic behavior of user's requirements; (3) architectural-centered implementation efforts of different POOTs.

To process these data quantitative metrics and expert-oriented estimation algorithms were elaborated, which are formalized and combined logically in a form of the proposed algorithmic model. Final complex estimation values of POOT effectiveness assessment are defined using the fuzzy logic method and the appropriate CASE-tool, which were successfully tested on real-life legacy software applications.

In future we are going to extend the collection of metrics for POOT-features assessment, and to apply some of alternative (to fuzzy logic method) approaches to final decision making support. Besides that it is supposed to continue our research with respect to the correlation issues between different kinds of CF and programming defects arose in an LSS source code.

References

1. Sommerville, I.: Software Engineering. Addison Wesley, Boston (2011)
2. Eilam, E.: Reversing: Secrets of Reverse Engineering. Wiley Publishing, Indianapolis (2005)
3. Apel, S. et al.: On the structure of crosscutting concerns: using aspects of collaboration? In: Workshop on Aspect-Oriented Product Line Engineering (2006)
4. Przyby³ek, 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, Reading (2004)
9. Gamma, E., et al.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (2001)
10. Sheldon, T., Jerath, K., Chung, H.: Metrics for maintainability of class inheritance hierarchies. *J. Softw. Maintenance Evol.* **14**, 1–14 (2002)
11. Harrison, R., Counsell, S.J.: The role of inheritance in the maintainability of object-oriented systems. In: Proceedings of ESCOM 1998, pp. 449–457 (1998)
12. Aversano, L., Cerulo, L., Di Penta, M.: Relating the evolution of design patterns and crosscutting concerns. In: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 180–192 (2007)
13. Hannemann, J., Kiczales, G.: Design pattern implementation in java and aspectJ. In: Proceedings of OOPSLA 2002, pp. 161–173 (2002)
14. Eaddy, M., et al.: Identifying, assigning, and quantifying crosscutting concerns. In: Workshop on Assessment of Contemporary Modularization Techniques (ACoMT 2007), Minneapolis, USA, pp. 212–217 (2007)
15. Filman, R., Elrad, S., Aksit, M.: Aspect-Oriented Software Development. Addison Wesley Professional, Reading (2004)
16. Figueiredo, E.: Concern-Oriented Heuristic Assessment of Design Stability. Ph.D. 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: International Workshop on Aspect-Oriented Programming ECOOP (1999)
19. Apel, S.: The role of features and aspects in software development. Ph.D. thesis, Otto-von-Guericke University Magdeburg (2007)
20. Tkachuk, M., Nagorny, K.: Towards effectiveness estimation of post object-oriented technologies in software maintenance. *J. Prob. Program.* **2-3**(special issue), 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. Eaddy, M., et al.: Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.* **34**(4), 497–515 (2008)
23. Aversano, L., et al.: Relationship between design patterns defects and crosscutting concern scattering degree. *IET Softw.* **3**(5), 395–409 (2009)
24. Walker, R., Rawal, S., Sillito, J.: Do crosscutting concerns cause modularity problems? In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE 2012), pp. 1–11 (2012)

25. Gottardi, T., et al.: Model-based reuse for crosscutting frameworks: assessing reuse and maintenance effort. *J. Softw. Eng. Res. Dev.* **1**, 1–34 (2013)
26. Tarr, P.L., et al.: N degrees of separation: multi-dimensional separation of concerns. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 107–119. ACM, Los Angeles (1999)
27. Official Web-site of System Thinking World community. <http://www.systems-thinking.org/kmgmt/kmgmt.htm>
28. Ramesh, K., Karunanidhi, P.: Literature survey on algorithmic and non-algorithmic models for software development effort estimation. *Int. J. Eng. Comput. Sci.* **2**(3), 623–632 (2013)
29. Tkachuk, M., Martinkus, I.: Model and tools for multi-dimensional approach to requirements behavior analysis. In: Kop, C. (ed.) *UNISON 2012. LNBIP*, vol. 137, pp. 191–198. Springer, Heidelberg (2013)
30. Saaty, T.L.: *Fundamentals of the Analytic Hierarchy Process*. RWS Publications, Pittsburgh (2000)
31. Garlan, D., Monroe, R., Wile, D.: ACME: an architecture description interchange language. In: *Proceedings of CASCON 1997*, Toronto, Canada, pp. 169–183 (1997)
32. Official Web-site of CIDE-project. http://www.witi.cs.uni-magdeburg.de/iti_db/research/cide/
33. Zadeh, L.A.: *Fuzzy Sets*. WorldSciBook (1976)
34. Official Web-site of IDEF Family of Methods. <http://www.idef.com>
35. Nagornyi, K.: Elaboration and usage of method for post object-oriented technologies effectiveness's assessment. *J. East-Eur. Adv. Technol.* **63**, 21–25 (2013)

Information and Communication Technologies in
Education, Research, and Industrial Applications
11th International Conference, ICTERI 2015, Lviv,
Ukraine, May 14-16, 2015, Revised Selected Papers
Yakovyna, V.; Mayr, H.C.; Nikitchenko, M.; Zholkevych,
G.; Spivakovsky, A.; Batsakis, S. (Eds.)
2016, XI, 157 p. 44 illus. in color., Softcover
ISBN: 978-3-319-30245-4