IADIS International Journal on Computer Science and Information Systems
Vol. 8, No. 1, pp. 47-65
ISSN: 1646-3692

UNDERSTANDING OF E-COMMERCE IS THROUGH FEATURE MODELS AND THEIR METRICS TO SUPPORT RE-MODULARIZATION

Kęstutis Valinčius, Vytautas Štuikys and Robertas Damaševičius. *Kaunas University of Technology, Software Engineering Department, Studentu str. 50, Kaunas, Lithuania*

ABSTRACT

The paper addresses the e-commerce system (interpreted here as information system - IS) understandability problem from the maintenance and evolution perspective. We propose a methodology that includes: (1) identification of the 3 *representative systems* (through clustering of a set of previously developed IS on the basis of user-oriented and temporal criteria); (2) extraction of feature models from representative systems for evaluation using similarity (measured by absolute difference value - ADV) and complexity metrics leading to the definition of the *reference IS* - RIS; (3) construction of the business logic feature model, represented as meta-graph, to reason about quality of the systems and to understand evolution trends; (4) feature model refactoring / modularization of the RIS; (5) Code-level modularization of the RIS. The methodology is supported by experiments to evaluate the analysed problems. The basic results are: (1) feature models of the representative systems; (2) metrics to evaluate complexity and similarity of the represented systems; (3) the re-engineered RIS to support better maintainability and design procedures.

KEYWORDS

Set of e-commerce systems, reference system, feature model, similarity and complexity measures, refactoring, modularization.

1. INTRODUCTION

A substantial part of software used in both large and small-to-medium enterprises are information systems for e-commerce (further IS). Such systems are seen as valuable assets for companies because they provide the underlying engines to improve processes for communication and to increase the overall market share. As it is observed by many reports (see, e.g., Lucca *et al*. 2006), such systems have been developed under the "pressure of short time-to-market and extremely high competition", thus without considering a sound

development methodology, their documentation is poor and, their quality is low and maintenance is difficult.

On the other hand, the user requirements to introduce new features during maintenance are constantly growing. Reasons for that are: users become more knowledgeable and are gradually better acquainted with functionality of the systems in the course of their exploitation, market pressure for new features, needs for better quality and performance in maintaining the systems. Therefore, introducing changes is a common practice. To respond to the challenges and to keep pace with arising new requirements, the systems are to be re-engineered, enhanced by new functionality. Also the improved maintenance procedures should be introduced. All these require a great deal of analysis and understanding of the IS.

The increasing number of companies wants to upgrade their systems by leveraging existing IT assets (such as business rules, data), minimizing costs and reducing time. The state-of-the-practice of software maintenance and evolution employs understanding, refactoring and re-engineering techniques that focus on code artefacts. However, recent advances have shifted the focus of evolution from the code level towards higher levels of abstraction and particularly the architectural and feature-based modelling levels (Trujillo et al., 2006; Liu et al., 2006). The ground behind this trend is that architecture and feature models capture the architectural knowledge of the IS, thus facilitate making new design decisions during evolution cycles, having full knowledge of past decisions.

In general, the aim of re-engineering is a predictable changeability management of the web-based legacy systems that cover the same application domain (e.g., e-commerce). Reasons for changeability management are: (a) to support maintainability of the systems; (b) to improve modularity of a system to making changes more easily; (c) to develop a new system on the basis of a legacy system; (d) to transform a given system by adding new functionality (e.g., extending e-commerce systems by c-commerce models). The kind of re-engineering when a designer tries to extract a higher–level representation (architecture, models, documentation) from the legacy code is known as reverse engineering (Gahalaut and Khandnor, 2010).

The principal aim of the paper is to present a methodology that, on the basis of analysis and understanding of a set of related IS through the relevant feature models and their evaluative characteristics (such as similarity, functionality and complexity grow, modularization level, etc.), enables to improve the development and maintenance activities. The *basic contribution* of the paper is the extension of the methodology (Valinčius et al., 2013) by adding two extra activities as follows: feature model refactoring/modularization and code-level re-modularization of the reference system defined as a result of previous analysis. The latter improvements enable to contribute to the further evolution and maintenance and to prepare for introducing *automatic generation* techniques in constructing new ISs.

The remaining part of the paper is organized as follows. Section 2 analyses the related work. Section 3 defines the terms and tasks. Section 4 presents the essence of the methodology. Section 5 describes evaluation of the approach and experiments. Section 6 provides conclusions.

2. RELATED WORK

The IS are emerging at the rate of tsunami as it is stated in reports (Lucca et al. 2006; Patel et al. 2007, Jung et al. 2011), where the following attributes are identified: (a) the development lacks of systematic approaches, (b) systems are kept running through a continual stream of patches; (c) systems suffer from low quality control and assurance; (d) systems have a poor structure; (e) quality of documentation is very low (if it exists at all). Good object-oriented practices have been almost entirely ignored in the traditional development of web applications. Most of these applications are incompatible with some of the most praised development and design techniques like modularization and abstraction (Halfaker, 2006). All these attributes are relevant to the set of IS we consider in the paper.

There are increasingly overlapping ideas in the areas of reverse engineering (RE), program understanding, refactoring, and model-driven development, all of which deal with program structure and maintenance or related with that (Batory, 2007). More details on feature modelling can be learnt from (Apel and Ch. Kästner, 2009; Ceri et al., 2007; Karam et al., 2008). The extensive analysis on program understanding is given in (Štuikys V. and Damaševičius, 2013, 174-178 p.). RE is a well-known common approach used in many fields of software engineering, such as software understanding, maintenance and evolution (Müller et al., 2000). The aim of RE is to improve understandability, maintainability and quality. Lopez-Herrejonet et al. (2011) identify eight refactoring patterns that describe how to extract the elements of features and to effectively modularize the features for the development of variable systems. *Features* are treated as increments in program functionality. Modularization also improves the program structure and has impact on understandability to introducing changes and, in this way, relates to refactoring (Shonle et al., 2008). Arzoky et al. (2012) describe the extended methodology of seeding to modularise sequential source code software versions and present modularization/similarity measures (including AVD – absolute value difference). Complexity metrics for programs are described in (Lehman et. al.1997; Damaševičius, 2009; Jung et al., 2011; Terceiro, et al., 2012) and for feature models in (Bagheri and Gasevic, 2011) from the maintainability perspective.

The following activity is the actual re-engineering. Refactoring is such a re-engineering activity in which the internal structure of a legacy system is changed without changing its external behaviour and functionality (Fowler et al., 1999). Partitioning the structure of the system using low-level dependencies in the source code, to improve the system's structure (Arzoky et al., 2011) according to anticipated objectives and requirements is also refactoring. Older monolithic systems, where modularization primarily involves splitting the monolithic code base into modules, for such newer systems which already have some basic modular structure, code decomposition is the only one of many possible activities (Rama and Patel, 2010). Software architectures commonly evolve into unmanageable mono-lights, leading to systems that are difficult to understand, maintain and evolve. In such common scenarios, developers usually have to invest considerable time in re-architecting the entire application, in order to restore its modular structure. However, re-architecting process is usually conducted in ad hoc way, without following any set of principled guidelines and methods (Terra et al., 2012). Periodic major restructuring of software applications at either the design or architectural level could be necessary (Raghvinder et al., 2008).

Software refactoring is a collection of re-engineering activities that aims to improve software quality too. Refactoring is commonly used in agile software processes to improve software quality after a significant software development or evolution (Shatnawi and Li, 2011). Traditional refactoring techniques have focused on the implementation stage, with source code as the primary artefact of the refactoring process. However, a recent trend is to apply the concepts of refactoring to higher levels of abstraction (Zhang et al., 2005). One general principle of powerful software systems is that they are built of many elements. Thus, when designing a system, the features of a system should be broken into relatively loosely bound groups of relatively closely bound features. The power comes from the interplay between the different elements (Volz et al., 2002).

Software evolution has largely been focused on low-level implementation artefacts through refactoring techniques rather than on the architectural level. However, code-centric evolution techniques have not managed to effectively solve the problems that software evolution entails (Avgeriou, 2006). The ability to develop components with identified common and variable parts, and rapidly instantiate product-specific versions is the key to many software product line approaches (Hutchesson and McDermid, 2011).

Unfortunately, refactoring requires understanding by an engineer both the techniques to be applied and the code to which they are applied to (Griffith et al., 2011).

We summarize analysis of this part to motivate our research as follows: 1) little-by-little web-based (e-commerce) systems degrade to a legacy code comprising a class of the modern legacy code; 2) there is ever-increasing need to improve the structure of the modern legacy code.

3. BASIC TERMS AND RESEARCH TASKS

IS for e-commerce is the system that supports B2C activities through the Internet. *Set of IS* – a family of related systems developed using the *same open source technology* (e.g., PHP) for the e-commerce. *Reference system* is the representative system having the most essential attributes of the family. *Feature* is "an externally visible characteristic" of the system or "an increment of program functionality" (see a survey of definitions in (Apel and Kästner, 2009)). *Feature model* is the representation of a system using feature-based notation (features, relationships among types of features and variant points and constraints). *Reverse engineering* (RE) is the process of extracting higher-level representation (e.g., models, etc.) of a system from its code (Patel, et al., 2007). *Understanding* of IS – a cognition process, based on extracted artefacts (feature models) through RE, to reason about the system functionality or aiming to perform some other activities such as change and redesign. *Reconfiguring* – the process of changing either component parameters within a system (such as colour, layout, input data, etc.) without changing component functionality. *Refactoring* – reducing the number of components (or both) aiming to adapt a system to the new context of use (Batory, D., 2007). *Modularisation* – the process of partitioning the structure of the software system into subsystems. Subsystems are clusters of source code resources with similar properties combined together to create a high-level attribute of the system. Modularisation also makes the problem at hand easier to understand as it reduces the amount of data needed by developers. According to Constantine and Yourdon a good modularisation of software systems leads to easier design, development, testing and maintenance [Arkozy]

We formulate the tasks as follows. Given a set of systems $S = \{s_i\}$, $i = [1, n]$ of the same application domain (e-commerce). The systems were developed in different time slots Δt between $[t_1, t_m]$ ($t_1 < t_i < t_m$), $\Delta t = t_{i+1} - t_i$, by different developers of the same organization, however, using the same technology. We need to identify a *reference system* among *S* to *understand evolution* of the entire family through obtained *feature models* and their characteristics such as *similarity* between the *reference system* and other *representative systems*, changes *in functionality and complexity* over time for future improvements. The set *S* is investigated under the following characteristics: $n = 40$, $m = 7$, $t_1 = 2005$ (year), $t_7 = 2011$ (year), $\Delta t = 1$ year, $n = n_1 + n_2 + \ldots + n_m = 2+3+4+6+10+8+7$, where n_i is the number of systems developed in time slot *i*. The task is formulated based on the following assumptions (hypotheses).

1. *The most general understanding of a system (systems) can be gained through categorization of its constituents (subsystems, objects, characteristics, etc.) and modelling.*
2. *The more systems within a given set are similar, the less effort to understand the entire set is needed.*
3. *Understanding of a set of related systems can be gained through the evaluation of their similarity, functionality and complexity growth.*
4. *Reference system feature model refactoring / modularization.*
5. *Code-level re-modularization of the reference system.*

4. GENERAL DESCRIPTION OF THE EXTENDED METHODOLOGY

The approach we have proposed to deal with re-engineering of the reference system can be seen as a Reverse Product Line Engineering (RPLE). Indeed, the traditional Product Line Engineering (PLE) starts at analysis of the domain to be implemented resulting in the identification the domain model (models) usually described at the high abstraction level (e.g., using feature-based notation); then the models serve for building the reference architecture for the family of the domain systems (Hutchesson and McDermid, 2011). The next process includes specification of the reference architecture leading to identification of components and generators (again represented at the higher-level of abstraction) to cover the essential artefacts of the whole domain. The above mentioned processes are called Domain Engineering (DE) (Falbo, 2002). The obtained high-level artefacts (models, architecture, components, etc.) are used as input to Application Engineering (AE) (Thurimella, 2011). The latter deals with the similar processes but under different aims and restrictions: 1) concrete requirements for each system to support a particular kind of products and 2) using the prescribed technology and relevant methodology to implement the DE artefacts as executable specifications to build PL systems.

To solve the formulated tasks, we have proposed the extended methodology. At the core of the methodology are the reverse engineering-based activities and feature-based modelling. The methodology consists of the following stages which in Figure 1 are described as a sequence of processes and models created by the adequate process. As it is difficult to extract from the given system the unique relevant model that could enable to consider and solve formulated tasks, we have split the analysis and modelling activities into stages. To represent the system models (obtained through reverse engineering), we have selected *feature models* (Apel and

Kästner, 2009) because they (a) describe the structure and functionality of a system at a higher-level of abstraction through entities known as features and (b) enable to simplify analysis to achieve the prescribed aims.

The aim of stage 1 is to reduce the search space for the identification of representative systems in order to select the reference system. The system developer analyses the systems using the available documents (e.g., initial requirements, code, users' opinion, and experts' comments) and performs clustering of systems as it will be explained in sub-section 4.1. The result of stage 1 is 3 representative systems (base, intermediate, and the latest system) as input information for the next stages.

The aim of stage 2 is to analyse the cluster of representative systems to identify the reference IS – RIS and to extract from it the feature-based models that specify the overall functionality. The process is based on reverse engineering. It includes mainly analysis of user interfaces though other system artefacts can be used too. As user interfaces are system dependent and, on contrary, feature-based models are system independent, we treat the latter as higher-level abstractions as compared to the first entities.

Stage 3 aims at extracting more details from the previously created model and to create the business logic feature model (BL FM) of the reference system as it will be explained in sub-section 4.2.

The aim of stage 4 is to reduce the scope of BL FMs transforming the latter into the more compact representation which we call meta-graph, the high-level model that provides information to understanding through RIS the set of ISs from the user and designer perspectives as it will be in sub-section 4.4. This model enables to evaluate the current structural characteristics and make the improvements through refactoring and re-modularization.

The aim of stage 5 is modularization of BL FM and meta-graph models to group and fragment RIS features into the hierarchical modular structure. The process defines, at a higher abstraction level, a unified modular structure of the architecture as it will be explained in sub-section 4.5.

The aim of stage 6 is the use of the initial RIS source code and modularized BL FM and meta-graph models to perform the code level re-modularization. Code-level modularization is oriented for the independent parent-child relationship between the APIs aiming to achieve the cohesion as low as possible.

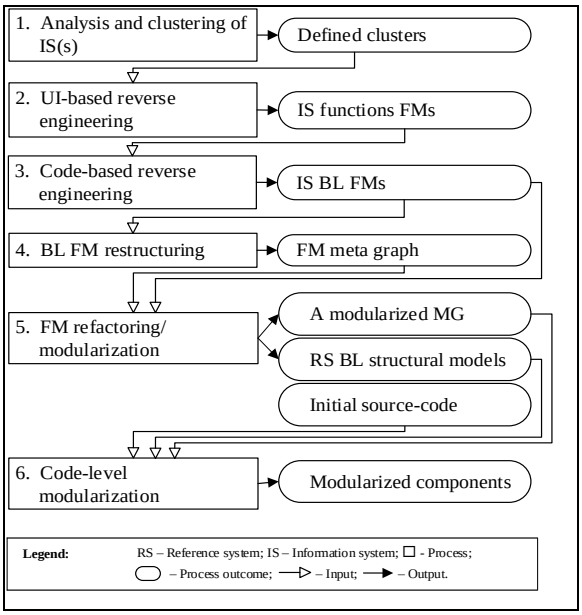

Figure 1. Re-engineering of e-commerce IS: a process-level view

4.1 Clustering of e-commerce Systems for Similarity Identification

Aim of analysis is first to select *representatives* due to the large number of systems (during the time slot $[t_1, t_m]$ of about 7 years, 40 systems were developed). To identify clusters of similar systems we used two criteria: 1) time slots of duration 1 year, when a particular system was developed and 2) user profile of the system. We motivate the criteria by heuristic observations obtained from the literature and experience of the developers. We identified 7 short time slots with the indication of the number of systems developed within each time slot (see task formulation) and 4 user profile types with respect to knowledge and experience of using IT products as follows: U_1 – novice users having used *the same or closely related* products and business rules (BRs); U_2 – novice users having used *slightly different* products and BRs; U_3 – experienced users having used *the same or closely related* products and BRs, and U_4 - experienced users having used *different products* and BRs.

To simplify the clustering problem, however, we found acceptable to admit only 2 large time clusters T_1 and T_2 (each being of 3.5 years duration see Figure 2) and 2 user profiles (U_{12} and U_{34}, meaning $U_{12} = U_1 \cup U_2$ and $U_{34} = U_3 \cup U_4$).

See the cluster identification results in Figure 2. Clusters C_1 ($U_{12} \times T_1$) and C_2 ($U_{12} \times T_2$) represent systems that were designed by reconfiguring the adequate ancestor system without *increasing* its functionality. All ancestors belong either to cluster C_3 ($U_{34} \times T_1$) or cluster C_4 ($U_{34} \times T_2$). The latter clusters represent systems derived sequentially within the specified time slots. Main properties of the systems are: 1) S_i is derived from S_{i-1} for all $i = [1, R]$ ($t_{i-1} < t_i$) and 2) $f(S_i) > f(S_{i-1})$, where $f(S)$ is functionality of the system S. The properties enable to draw the evolution curve denoted in bold in Figure 2 and meaning the growth of functionality. It is

clear that systems on the line are similar because they have the base functionality inherited from the base S_0 and some extra functionality added in the course of evolution.

Note that clusters C_1 and C_2 were neglected because their systems do not increase the functionality. Due to the system similarity (we will evaluate its degree later), it is enough to consider only some systems along the evolution curve. We have selected 3 systems as the most representative ones: S_0 (base), S_3 (intermediate system at the boundary of T_1 and T_2) and S_R (having the largest functionality). The latter has been identified as a *reference system* (note that we use S_R in the formal notation and RIS in informal reasoning). Its role is twofold: (a) it enables to track functionality of the existing systems to support their maintenance; (b) it serves as a sound template to provide basic components for generalization in order to support automatic changeability in designing new systems in the future.

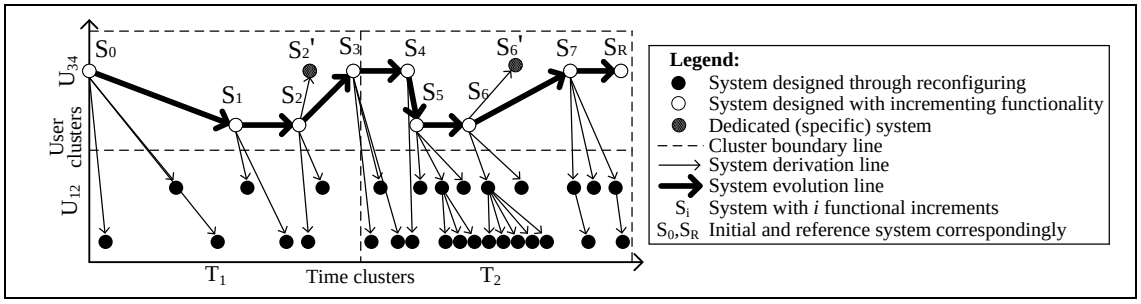

Figure 2. System clusters to define the representative systems (designer's view)

4.2 Extraction of Feature Models from Representative Systems

The aim of analysis at stage 2 and 3 is first to build models of the representative systems S_0, S_3 and S_R, and then to identify the degree of the *model similarity* and to evaluate complexity of the models. To represent system models, we have selected feature models because they (a) describe the structure and functionality of a system at a higher-level of abstraction through entities known as features and (b) enable to simplify analysis to achieve the prescribed aims. We introduce 4 views to our systems aiming to simplify analysis as follows:

$$f(S) = f^{(u)}(S) \cup f^{(m)}(S) \cup f^{(p)}(S), \qquad (1)$$

where $f^{(u)}(S)$, $f^{(m)}(S)$, $f^{(p)}(S)$ is *the user-based, the maintainer-based* and the *system provider-based functionality of S*, respectively; $f(S)$ – the total functionality of the system from the *designer's perspective S.*

As it is supposed that $f^{(u)}(S) \subset f^{(m)}(S)$, $f^{(u)}(S) \subset f^{(p)}(S)$ meaning that user-based functionality is also taken into account in the remaining views, we are able to model representative systems considering the only its main part, that is, the user-based functionality $f^{(u)}(S)$. Further, by feature models (FM) (either functional (F FM) or business logic (BL FM)), we mean the models constructed to represent the $f^{(u)}(S)$ view only.

Below we apply RE as a sequence of steps at stage 2 and 3 (Figure 1) to extract their feature models.

1. The base system S_0 is selected first (it represents the root of a feature model) and its UIs are navigated multiple times from the highest level interface to the lowest one. The item (UI elements) within any UI is treated as a functional feature. If this item must be selected

always, it is treated as a *mandatory feature* (denoted as black circle, see Figure 3), otherwise it is treated as an *optional feature* (denoted as white circle). Usually, the lowest-level UI represents a *variant point* with variants of alternative features as grouped features.

2. The navigation process is repeated in order to cover all paths by selecting the remaining functional items within each UI. Other features are extracted as it is described by step 1 and represented as the parent-child relationship tree (sub-tree).
3. The constraints of the type *require* or *exclude* (if any) are identified among variants, variant points or intermediate features. This activity is based on the knowledge of the analyser (usually he/she is a designer of the system). The parent-child feature relationship tree combined with constraints is the functional FM (F FM) of S_0.
4. The system S_3 is analysed next in a similar way having the F FM of S_0 as a basis for the S_3 F FM. This means that we need to add to the obtained model the only new features that appear in UIs of S_3.
5. Finally, the system S_R is analysed (having in mind the F FM of S_3 as the basis), but now adding the only new features from the UIs of S_R, as seen in Figure 3.
6. The BL FMs for the representative systems are constructed on the basis of F FMs by adding business logic features to the F FMs. The analyst needs to work partially at the code level in order to extract the implementation related knowledge such as modules and APIs. The BL FM (FMs) (an extract of the model is given in Figure 4) serves for two purposes: 1) to evaluate the systems by model evaluation metrics (see Section 4.3) and 2) to construct meta-graph for the S_R to understand it and the entire IS family from the user and designer perspective.

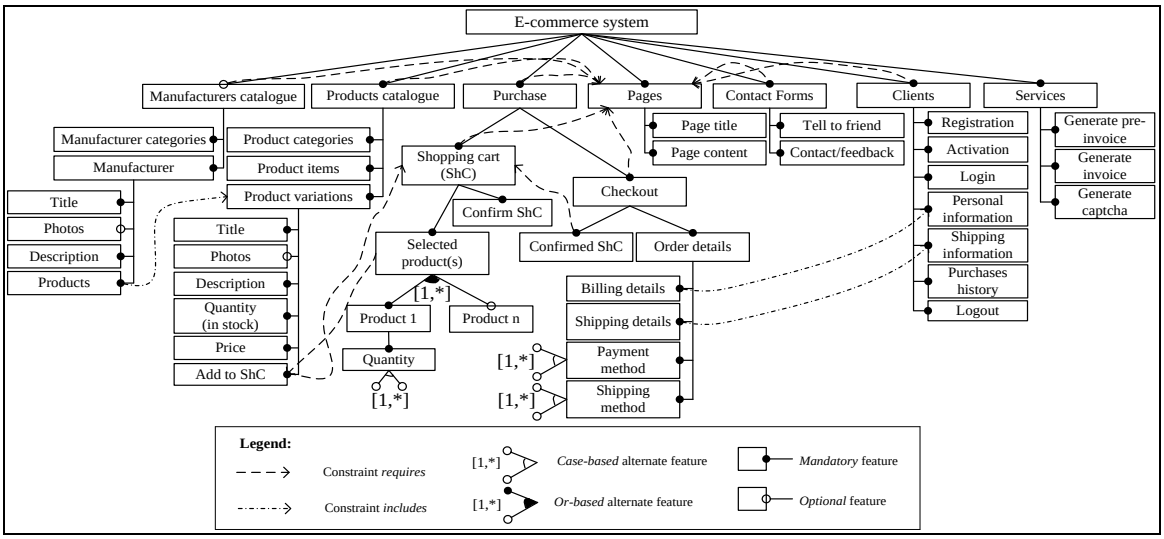

Figure 3. Fragment of feature-based reference architecture at function level

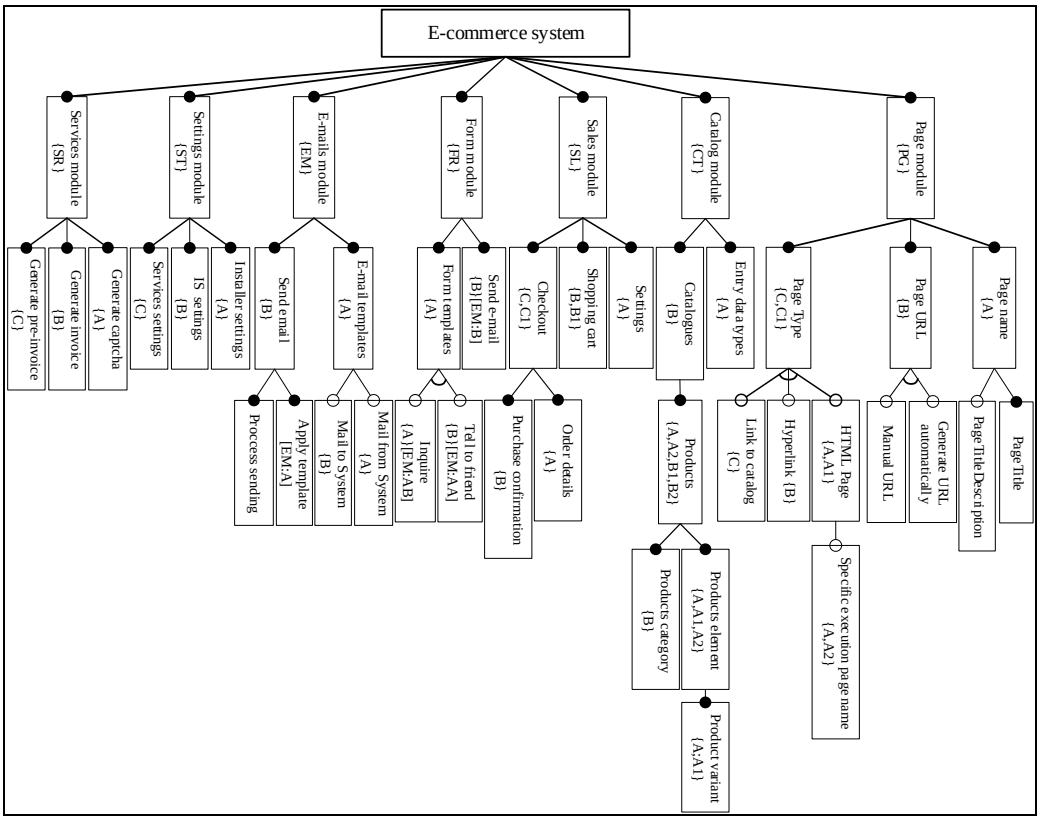

Figure 4. RIS fragment of feature-based model at the business logic (BL) level

4.3 Metrics to Evaluate Feature Models

We introduce a feature *model similarity metric* (Eq. 2) expressed as the *absolute value difference* (AVD) adopted from (Arzoky et al., 2012). The obtained models are then compared and evaluated using the metric.

$$AVD(X,Y) = \sum_{i=1}^{s}\sum_{j=1}^{s}\left|x_{ij} - y_{ij}\right| \qquad (2)$$

Where X and Y are binary *feature* matrices of two comparable feature models for $S^{(u)}$; x_{ij} , y_{ij} – elements of the matrices; s – is the number of features of the largest matrix. x_{ij} , y_{ij} =1, if features i and j have the relationship or constraint branch; otherwise x_{ij} , y_{ij} = 0. Note that the feature model with a smaller number of features is supplemented by additional void features (isolated nodes without branches) in order to equalize the size of both matrices.

To evaluate complexity of FMs, we use two measures (Štuikys and Damaševičius, 2013, see pages 213-216): *cognitive complexity*, which is calculated as the maximal number of

feature levels in the hierarchy or the maximal number of features in each level, and the *compound complexity* (estimated by Eq. 3):

$$C_m = F^2 + (R_{and}^2 + 2R_{or}^2 + 3R_{case}^2 + 3R_c^2)/9, \qquad (3)$$

C_m - compound complexity measure; F, R_{and}, R_{or}, R_{case}, R_c – the number of {all features, mandatory relationships, optional relationships, alternative relationships, relationships among variants including constraints}, respectively; the division coefficient is for equalizing the role of relationships. We present and analyse the estimated values obtained using Eq. 2 and Eq. 3 in Section 5.

4.4 Meta-graph as a Model to Understand Business Processes

The BL FM provides essential attributes to understand system functionality by designer well; however, the model is less helpful for understanding the user's requirements because it may be too complicated for the user. For example, when the designer interacts with a user aiming to know his/her requirements as compared to those that are implemented into the already existing IS, all details of the model are not needed. On the other hand, the designer needs to communicate with the user using his/her language. Though feature names are usually expressed in the user-understandable fashion, the FM should be re-factored and reduced. We introduce yet another (higher) level within FM; and we construct the higher-level model, called *meta-graph* (Figure 5). The intention of the meta-graph is to specify sub-processes within BL expressed through sequences of features needed to perform the BL operations. The meta-graph $G(X, (E, U^w))$ notation we have adopted from (Basu and Blanning, 2007) and apply it in our context as follows. Two nodes, denoted as x_0 and x_t ($x_0 \neq x_t$; $x_0, x_t \in X$), represent the initial and final states respectively. All sub-processes begin at *Start* state and terminate at the *End* state (if the process that consists of a set of sub-processes is complete). A node $x_i \in X$ ($i = [1..t-1]$) represents the business sub-process. There are weighted and non-weights nodes (note that here we found the necessity of changing weighting of arcs by nodes of the meta-graph as compared to (Valincius et al., 2013)). The arcs $u_{ij} = (x_i, x_j)$ ($u_{ij} \in U$, $i, j \in [0..t]$) represent the execution sequence of sub-processes and the node weight w_{ij} represents a compound structure of features taken from Eq. 4, using BNF-like notation.

$$w_{ij} = \{<number\_of\_BL\_features>\ ";"\ \{[<FM\_class\_ID>\ ":"\ [<set\_\ of\ \_class>]\ ";"\ ]\}\ ";"\ <LOC>\}, \qquad (4)$$

where <*number_of_BL_features*> is the total number of features in the BL FM that are to be selected to execute a sub-process (e.g., the sub-process "Order details confirmation" requires 38, see Figure 5);

<*FM_class_ID*> :<*set_ of _class*> is the BL FM category identifier with a set of features within each category (e.g., the sub-process "Order details confirmation" contains two categories of features *PG* and *SL*; the first has only one set of features (*CAA4*); the second category *SL* has 1 set of features (*CA*).

< *LOC*> - the number of code lines to implement the sub-process (e.g., 3057 for the same sub-process).

[<*x*>] – list of <*x*> items.

The set *E* is the *feedback arcs* to define the sequence of sub-processes which return to the *Start* state in order to complete some task. Formally, the set is defined as: $E = \{\forall_{k\in[1,t-1]}(x_k, x_0)\}$, i.e. each sub-process has a feedback arc for returning to the initial state, if there is the need for terminating the sub-process (e.g., if the sub-process "Products catalogue" was executed to see the product category list only).

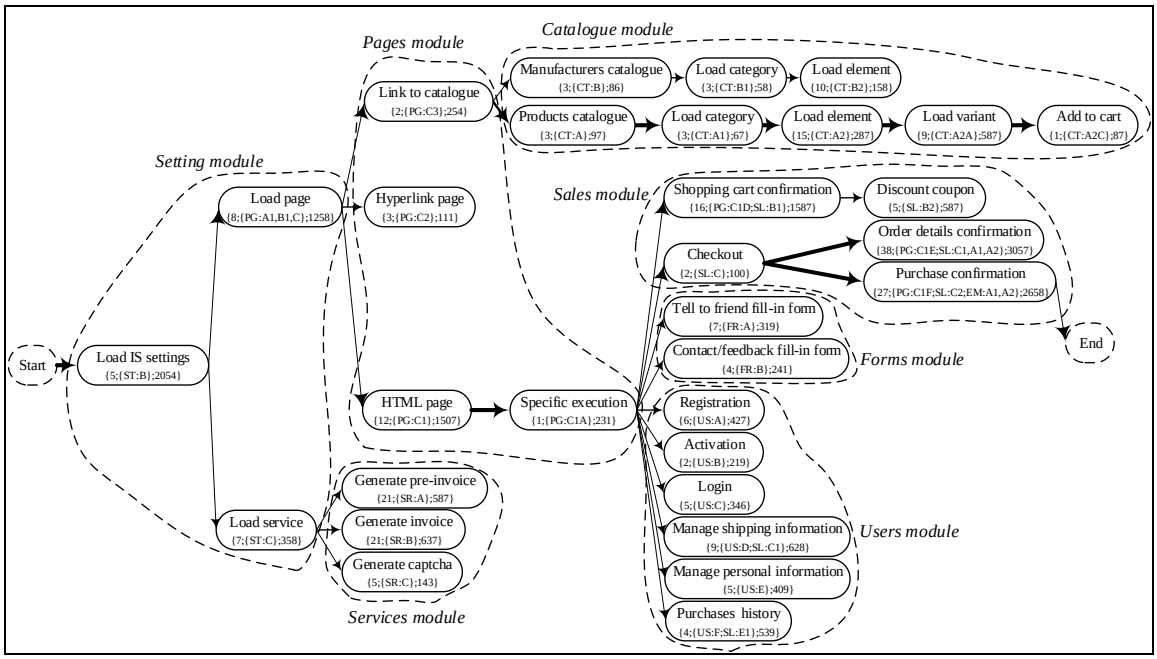

Figure 5. Modularized meta-graph for understanding "Product purchase" task

4.5 Re-engineering of Reference System through Re-Modularization

We have also identified some specifics in using feature models in our approach as compared to the processes of the development of PLE. If feature modelling as applied to the development systems are usually based on some uncertainty, hypothesis and anticipation in analysis, on the contrary, feature modelling of legacy systems are based on the really existing features. The main focus is to recognize, to understand and to extract the features and their relationships. The modularization task we consider in our approach is very similar to the formation of components by selecting and grouping the relevant features when designing and representing components at the high-level of abstraction. Modularization as applied to components of a legacy system may require feature splitting that results in the possible appearance of the same feature in different components. The latter may be influential also to constraints. We employ the notion of API (Application Programming Interface) as the basis for our structural metrics (Sarkar et al., 2007).

We perform refactoring through re-modularization at two levels: feature models and code. By applying refactoring at the model level, we have made the transition more systematic and less error-prone (Rossi et al., 2008). The initial RIS was poorly structured, consisting of only 5

major modules with poor API techniques. The restructuration firstly involved grouping and de-fragmentation of features later, thus allowing to constructing the hierarchical structure (see Figure 6). However, object-oriented software is harder to change than it might appear to be at first. Changing an object-oriented system often requires changing the abstractions embodied in existing object classes and the relationships among those classes. This involves structural changes such as moving variables and functions between classes and partitioning a complicated class into several classes (Opdyke, 1992).

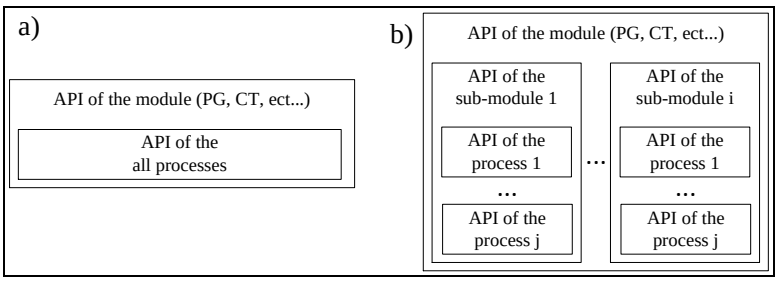

Figure 6. Simplified architecture-level structure of RIS before (a) and after (b) refactoring/modularization

Unified API structure was defined (see Figure 7) using best practises and consist of 5 inner components: 1) sub API's definitions, 2) initializer, 3) engine were all business logic is placed (spitted into locale independent language translations, business logic core and visualisation templates), 4) libraries for external components handling and 5) resources for multimedia information handling (visual layout information, real-time processing scripts and graphical or multimedia information storing)

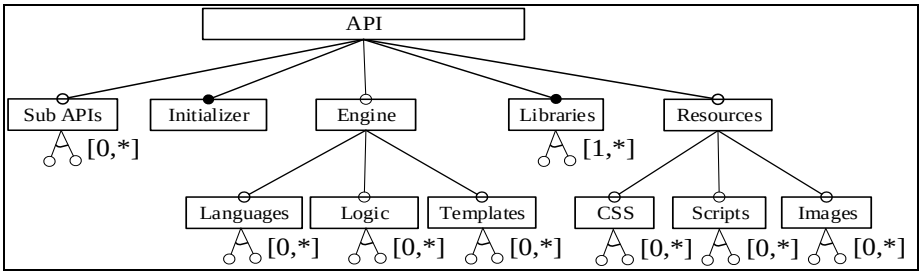

Figure 7. Unified feature model for any level component of Figure 6 b

5. EVALUATION OF THE APPROACH AND EXPERIMENTS

If the conventional PLE approach is well suited for the development of new PL systems, the reverse PLE we considered is better suited for maintenance and evolution of legacy systems. As there are evident signs of blurring boundaries between the software system development and evolution (Chikofsky, 1990), the proposed approach can be included into the processes of Round Trip Engineering (Ciccozzi, 2013). There are some restrictions in using RPLE. The approach is applicable to the systems which are homogeneous in terms of the used technology

and application domains. There is also the lack of suitable tools that support analysis, understanding and transformations (especially reverse transformation) of legacy systems (Mealy, 2006; Katić, 2009; Mens and Tourwe, 2004).

We present the algorithm to solve the main task (*Product purchase*) using the constructed meta-graph. The algorithm models the task solving through the identification of series of routes within the meta-graph (see bold branches in Figure 5) as follows:

1) From *Start* to *Add to cart* & return to *Start*; 2) From *Start* to *Shopping cart confirmation* & return to *Start*; 3) From *Start* to *Order details confirmation* & return to *Start*; 4) From *Start* to *Purchase confirmation* & to *End*.

The algorithm is simple enough to explain the meaning of business processes for the novice user of IS. If the user wants new features to be added to his/her IS, the model and the algorithm is helpful too, because it points to a particular part of FM to explain possible extensions of the system (i.e., to elicit new requirements). The model provides the designer with the extremely useful information to track FMs to introducing changes into code. Also, the designer is able to reason about the level of quality of the previously developed IS. Furthermore the model provides some information on modularization (it is clearly seen that the sub-processes, as a source for modularization, have a quite different number of features, meaning the *size problem of modules;* it is known that to support changeability, modules of the system should be approximately equal in size). The model is also beneficial to deal with the so-called *concept location problem* because designer is able to see features at high abstraction level, and then, to navigate through different levels of abstraction to select the needed code.

The results of experiments we have carried out also contribute to the understanding of the systems via the identified changes in functional similarity, and complexity growth. Three representative systems (S_0, S_3, S_R identified as the initial, intermediate and the latest), within 7 years of their evolution, have been selected for investigation. Note that only the essential part of a system (identified as the user-based vision of a system) was used in the experiments (due to simplification of the process). We have identified the similarity and functionality changes of systems over the evolution period as compared to the initial system. Using AVD as a similarity measure (see Figure 4), we estimated (for the BL FM only) that the similarity evolved roughly linearly, though the number of systems delivered in the second half (3.5 years) of the total period was much higher. Figure 8 (see the right side) also provides with information on the code complexity changes (growth) that were measured by LOC. Again, only the essential part of representative systems was taken into account.

Evaluation of system complexity has been estimated at the model level, too. Results are summarized in Table 1, where several model complexity measures, such as cognitive complexity, structural and compound complexity, are given to identify the growth of complexity over time (2nd Lehman's Law).

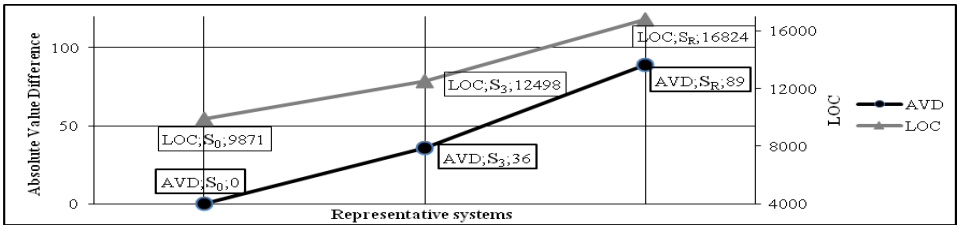

Figure 8. Similarity and simplified BL FM complexity increase

Table 1. Complexity evaluation of e-commerce systems feature models at business-logic level

Representative Systems	FM complexity measures			
	# of VPs	Cognitive complexity	Structural complexity	Compound complexity
S_0	40	8	98	10 174
S_3	49	8	123	15 811
S_R	65	9	161	59 332

The benefits of models are: 1) reference system models are representative items of the entire IS family enable to reducing the space of variants in understanding them; 2) models cover the underlying functional attributes of the family; 3) similarity measures enable to observe the evolution of systems functionality growth and track the introduced changes over time; 4) meta-graphs help to elicit new requirements.

The RS had a poor module oriented structure and was based on strong cohesion and no defined interfaces. Re-engineering allowed specifying and grouping the functional blocks where in re-modularization phase they were specified as independent components (APIs). The RS structure of 5 modules was fragmented to 45 unified API structures (see Figure 9).

Evaluation of system modularization has been estimated at the API-level. Results are summarized in Table 2, where several structural measures are given to identify the structural growth after re-modularization.

Table 2. Structural re-modularization evaluation of e-commerce system

Modules & sub-modules of the system	Structural characteristics of the system			
	Before		After	
	# of API's	Size (# of Features/LOC)	# of API's	Size (# of Features/LOC)
Settings {ST}	1	33 / 4270	3	33 / 4467
Page {PG}	1	31 / 2587	3	31 / 2797
Catalogue {CT}			12	61 / 2249
Products	1	61 / 1570	8	28 / 1562
Manufacturers			3	8 / 687
Sale {SL}			10	88 / 8512
Shopping cart			3	14 / 2097
Order details	1	88 / 7597	3	32 / 3181
Purchase confirmation			2	10 / 2637
Order information			1	8 / 597
User {US}			10	52 / 3472
UserRALP	1	52 / 2568	7	12 / 2681
UserSales			2	30 / 791
Form {FR}	0	16 / 560	2	16 / 827
Services {SR}	0	35 / 1367	4	35 / 1719

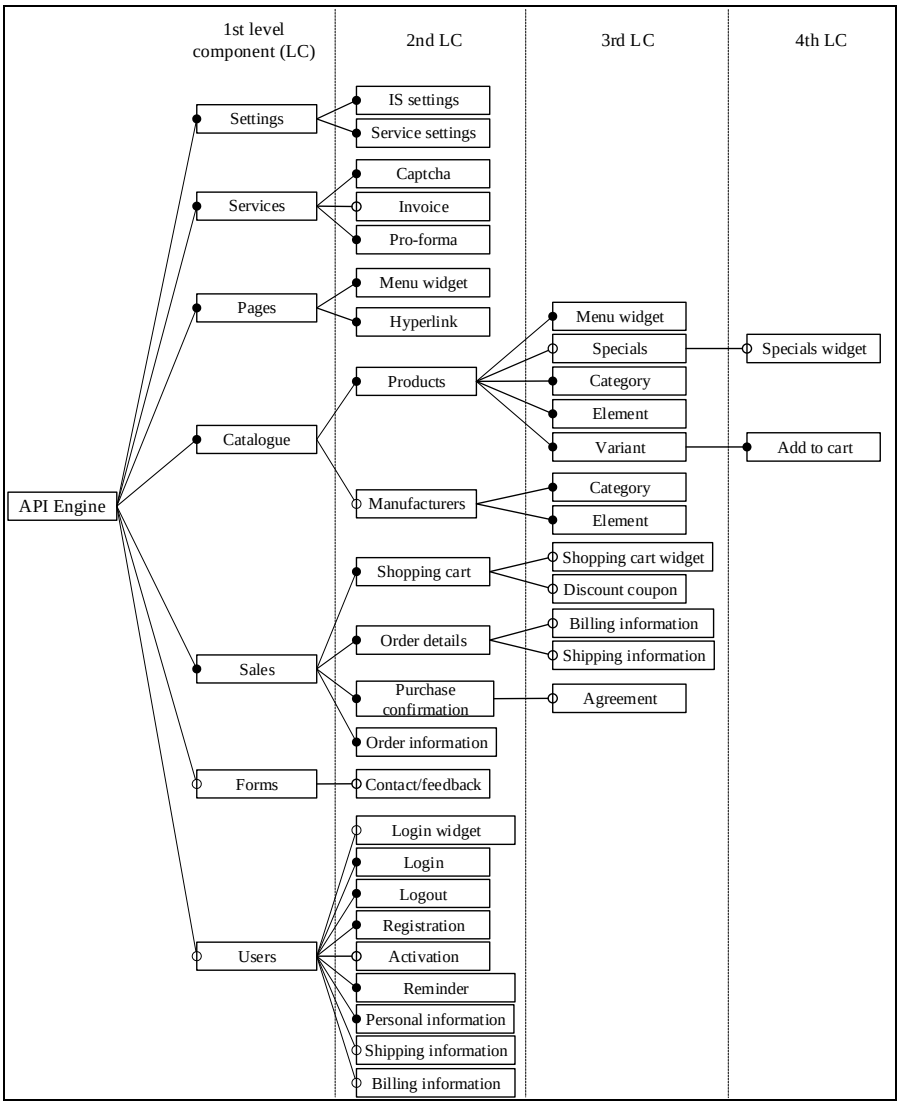

Figure 9. Re-modularized RIS structure at the API-based architecture-level

6. CONCLUSIONS

As e-commerce systems (for small-to-medium enterprises) are evolving extremely rapidly, their maintenance and evolution tasks are complex and require essential effort to analyse and understand them. The understanding problem we have analysed is the primary step to improve maintainability of such systems. The next step is refactoring and modularization. Those activities enable to achieve two important outcomes: 1) to improve changeability/

maintainability and 2) to create the opportunity to generalize components for design automation due to improved modularization. The models we have created using the methodology are beneficial for all actors involved in the process (including users trying to transfer their requirements for system innovations), though to the different degree. Though the methodology has been devised using a specific set of IS, we hope that it might be useful for a broader kind of distributed systems.

ACKNOWLEDGEMENTS

The work described in this paper has been carried out partially within the framework the Operational Programme for the Development of Human Resources 2007-2013 of Lithuania „Strengthening of capacities of researchers and scientists" project VP1-3.1-ŠMM-08-K-01-018 „Research and development of Internet technologies and their infrastructure for smart environments of things and services" (2012- 2015), funded by the European Social Fund (ESF).

REFERENCES

Apel, S. and Kästner, Ch., 2009. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, Vol. 8, No. 5, July–August, p.p. 49-84.

Avgeriou, P., and Zdun, U., 2006. Architecture-Centric Evolution: New Issues and Trends. Object-Oriented Technology. ECOOP 2006 Workshop Reader. Springer Berlin Heidelberg, pp. 97-105.

Arzoky, M. et al., 2012. A Seeded Search for the Modularisation of Sequential Software Versions. In *J. of Object Tech.*, vol. 11, no. 2, p.p. 1–22.

Arzoky, M. et al., 2011. Munch: An Efficient Modularisation Strategy to Assess the Degree of Refactoring on Sequential Source Code Checkings. IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, Germany, pp. 422-429.

Bagheri, E. and Gasevic, D., 2011. Assessing the maintainability of software product line feature models using structural metrics. *Software Qual. Journal*, Springer, p.p. 579-612

Basu, A. and Blanning, R.W., 2007. *Metagraphs and their applications*. Springer, Printed in USA.

Batory, D., 2007. Program Refactoring, Program Synthesis, and Model-Driven Development. *Invited Presentation at European Joint Conf. on Software Theory and Practice of Software (ETAPS)*, Compiler Construction Conference.

Ceri, S. et al., 2007. Model-driven Development of Context-Aware Web Applications. *Journal ACM Transactions on Internet Technology (TOIT)*, ACM, NY, USA, Vol. 7, Issue 1.

Chikofsky, E. and Cross, J., 1990. Reverse engineering and design recovery: A taxonomy. IEEE Software, Vol. 7, No. 1, pp. 13–17.

Ciccozzi, F., 2013. Round-trip support for extra-functional property management in model-driven engineering of embedded systems. Information and Software Technology. Vol. 55, No. 6, pp. 1085-1100.

Damaševičius, R., 2009. Visualization and Analysis of Open Source Software Evolution using an Evolution Curve Method. *In Databases and Information Systems V*, Eds. H-M. Haav and A. Kalja, IOS Press, p.p. 205-216.

Falbo, A.R. et al, 2002. An ontological approach to domain engineering. Proceedings of the 14th international conference on Software engineering and knowledge engineering (SEKE '02). New York, USA, pp. 351-358.

Fowler, M. et al., 1999. Refactoring: Improving the Design of Existing Code, Addison Wesley Longman Inc.

Gahalaut, K, A. and Khandnor, P, 2010. Reverse Engineering: an Essence for Software Re-engineering and program analysis. International Journal of Engineering Science and Technology, Vol. 2, No. 6, pp. 2296-2303.

Griffith, I. et al., 2011. Evolution of Legacy System Comprehensibility through Automated Refactoring. International Workshop on Machine Learning Technologies in Software Engineering, Lawrence, Kansas, USA, pp. 35-42.

Grubb, P and Takang, A.A., 2007. *Software maintenance (concepts and practice)*, World scientific, London.

Halfaker A., 2006. Modular Application Framework for Web Applications. Proceedings 39th Midwest Instruction and Computing Symposium, Wisconsin, USA.

Hutchesson, S., and McDermid, J., 2011. Towards Cost-Effective High-Assurance Software Product Lines: The Need for Property-Preserving Transformations. Software Product Line Conference (SPLC), 2011 15th International, Munich, Germany, pp. 55–65.

Jung, W.-S. et al., 2011. An Entropy-Based Complexity Measure for Web Applications Using Structural Information. *Journal of Information Science and Engineering*, 27, p.p. 595-619.

Karam, M. et al., 2008. A product-line architecture for web service-based visual composition of web applications. *Journal of Systems and Software*, Elsevier Science Inc. NY, USA, Vol. 81, Issue 6, p.p. 855–867.

Katić, M., and Fertalj, K., 2009. Towards an Appropriate Software Refactoring Tool Support. Proceedings of the 9th WSEAS International Conference on APPLIED COMPUTER SCIENCE. Genova, Italy, pp. 140-145.

Lehman, M.M., 1997. Metrics and Laws of Software Evolution - The Nineties View. *IEEE METRICS*, p.p. 20-32.

Liu, J. et al., 2006. Feature oriented refactoring of legacy applications. In ICSE '06: *Proceeding of the 28th international conference on Software engineering*, NY, USA, p.p. 112–121.

Lopez-Herrejon, R. E. et al., 2011. From Requirements to Features: An Exploratory Study of Feature-Oriented Refactoring. *Proc. of 15th International Software Product Line Conference*, p.p. 181-190.

Lucca, D. et al., 2004. *Reverse Engineering Web Applications: The WARE Approach*. Journal of Software Maintenance and Evolution, Research and Practice, Vol. 16, p.p. 71-101.

Mealy, E. and Strooper, P., 2006. Evaluating software refactoring tool support. Proceedings of the Australian Software Engineering Conference (ASWEC), Sydney, Australia, pp. 331–340.

Mens, T. and Tourwe, T., 2004. A Survey of Software Refactoring. IEEE Transactions on Software Engineering, Vol. 30, No. 2, pp. 126–139.

Muller,H.A. et al., 2000. Reverse Engineering: A Roadmap. *Proceeding ICSE '00 Proceedings of the Conference on The Future of Software Engineering*, ACM New York, NY, USA, 2000, p.p. 47–60.

Opdyke, F. W., 1992. Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks. PhD, University of Illinois at Urbana-Champaign.

Patel, R. et al., 2007. Reverse Engineering of Web Applications: A Technical Review. *Reverse Engineering of Web Applications: A Technical Report.* The University of Liverpool.

Raghvinder, S. et al., 2008. Structural Epochs in the Complexity of Software over Time. IEEE Software, July/August 2008, pp. 66-73.

Rama, G. and Patel, N., 2010. Software modularization operators. In 26th International Conference on Software Maintenance (ICSM), Timisoara, Romania, pp. 1–10.

Rossi, G. et al., 2008. Refactoring to Rich Internet Applications. A Model-Driven Approach. Proceedings of the Eighth International Conference on Web Engineering (ICWE '08), IEEE Computer Society Washington, DC, USA, pp. 1–12.

Sarkar, S. et al., 2007. API-Based and Information-Theoretic Metrics for Measuring the Quality of Software Modularization. IEEE Transactions on Software Engineering, Vol. 33, No. 1, pp. 15-33.

Shatnawi, R. and Li, W., 2011. An Empirical Assessment of Refactoring Impact on Software Quality Using a Hierarchical Quality Model. International Journal of Software Engineering and Its Applications, SERSC Korea, Vol. 5, No. 4, pp. 127–150.

Shonle, M. et al., 2008. When Refactoring Acts like Modularity. Keeping Options Open with Persistent Condition Checking, *WRT'08*, October 19, 2008, Nashville, Tennessee, USA, ACM.

Štuikys, V. and Damaševičius, R., 2013. *Meta-programming and Model-Driven Meta-Program Development (Principles, Processes and Techniques)*, Springer-Verlag London ISBN 978-1-4471-4125-9.

Terceiro, A. et al., 2012. Understanding Structural Complexity Evolution: A Quantitative Analysis. Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on Computing & Processing (Hardware/Software), p.p. 85–94.

Terra, R. et al., 2012. An Approach for Extracting Modules from Monolithic Software Architectures. IX Workshop de Manutencao de Software Moderna (WMSWM). Fortaleza, Brazil, pp. 1-8.

Thurimella, K.A., 2011. On the communication problem between domain engineering and application engineering: formalism using sets, conflicts of-interests and artifact redundancies. ACM SIGSOFT Software Engineering Notes. Vol. 36, No. 4, pp. 1-5.

Trujillo, S. et al., 2006. Feature Refactoring a Multi-Representation Program into a Product Line. *GPCE '06 Proc. of the 5th Intern. Conf. on Generative programming and component engineering*, ACM, NY, USA, p.p.191–200.

Valinčius, K. et al., 2013. Understanding of E-Commerce IS Through Feature Models and Their Metrics. International Conference Information Systems 2013, Lisbon, Portugal, pp. 55-62 (Awarded by Certificate of the Best Quantitative Research Paper).

Volz, R. et al., 2002. Towards a Modularized Semantic Web. Proceedings of the ECAI-02 Workshop on Ontologies and Semantic Interoperability, Lyon, France, Vol. 64.

Zhang, J. et al., 2005. Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. Model-driven Software Development, Springer.