Cent. Eur. J. Comp. Sci. • 2(3) • 2012 • 300-315
DOI: 10.2478/s13537-012-0020-x

VERSITA

Central European Journal of **Computer Science**

Analyzing stereotypes of creating Graphical User Interfaces

Review Article

Michaela Bačíková*, Jaroslav Porubän†

Department of Computers and Informatics,
Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 04200 Košice, Slovakia

Received 31 January 2012; accepted 17 August 2012

Abstract: A graphical user interface (GUI, UI) is an important part of an application, with which users interact directly. It should be implemented in the best way with respect to understandability. If a user does not understand the terms in the UI, he or she cannot work with it; then the whole system is worthless. In order to serve well the UI should contain domain-specific terms and describe domain-specific processes. It is the primary source for domain analysis right after domain users and experts. Our general goal is to propose a method for an automatic domain analysis of user interfaces. First, however, the basic principles and stereotypes must be defined that are used when creating user interfaces and rules must be derived for creating an information extracting algorithm. In this paper these stereotypes are listed and analyzed and a set of rules for extracting domain information is created. A taxonomy of UIs and a taxonomy of components based on their domain-specific information is also proposed. Our DEAL method for extracting this information is outlined and a prototype of DEAL is presented. Also our goals for the future are listed: expanding the prototype for different components and different types of UIs.

Keywords: domain analysis • graphical user interface • components • reflection • aspect-oriented programming

1. Introduction

Domain analysis (DA) [12] is a process that is currently most often used in the software systems design and analysis. Fig. 1 describes the most common process of domain analysis. It is necessary to collect information from various sources: a) domain users and experts; b) existing documents; and c) existing software systems. After that, the collected information is categorized, analyzed and some form of a domain model is created. Domain models have many forms. The most commonly used is the FODA notation [13]. FODA method analyzes product lines and resulting domain model contains varying/consistent and mandatory/optional features in a domain and it also defines the vocabulary used in the domain, concepts, ideas and phenomena within the system. Then this model (often with a connection with generators and libraries of reusable components or frameworks) is used for creating a new software system or editing an existing system. The domain analysis is often performed by a **domain analyst**.

* *E-mail: michaela.bacikova@tuke.sk (Corresponding author)*
† *E-mail: jaroslav.poruban@tuke.sk*

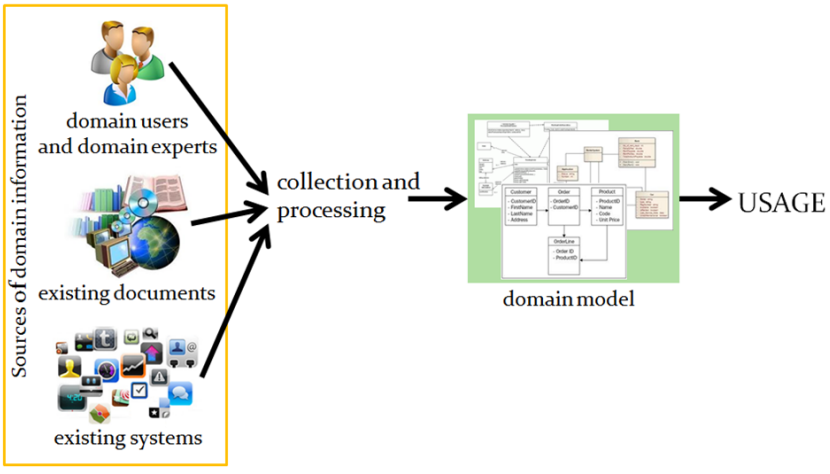

Figure 1. The process of domain analysis.

1.1. Tasks and goals

The general goal of our research is to **propose a method for an automatic analysis of a software system and a synthesis of a domain model**. The **input** of such method will be a software system with a graphical user interface constructed of components and the **output** of the method will be a (semi-)formal domain model.
The method will use a dynamic analysis of **user interfaces**. The analysis will be implemented using *reflection* and *aspect oriented approach*. The main processes of such a method include: i) an analysis of UI *components* and their *relations*; ii) a synthesis for domain *terms* descriptions and their *relations*; iii) an analysis of *event sequences* in a UI; and iv) a synthesis for *domain process descriptions*. The final aim is to implement a prototype using this method and verify it on a set of software systems.
A presumption for proposing such a method is to *identify and list the most common stereotypes of creating GUIs* considering domain terms and processes. This list of stereotypes will serve for deriving rules for extraction of domain information.
Therefore the **main goals of this paper** are:

- to identify and list the most common stereotypes of creating GUIs,
- to analyze the stereotypes with respect to domain analysis,
- to design and present a pseudo-code of the domain analysis algorithm.

1.2. Problems of existing approaches

Modelling and creating domain models from gained information is now widely supported by numerous methodologies and tools. Gaps remain especially in the area of data collection. Problems of existing approaches in this area can be divided into three groups based on the information source:

(a) Data collection ***from domain users and domain experts*** is most often carried out through reviews, questionnaires and forms. These methods are often time consuming and require both the willingness of users and experts and a certain level of skill at the side of the domain analyst. On one hand, a user does not always have time or mood to talk with the domain analyst or to fill out some questionnaires and forms. The domain analyst on the other hand must be able to ask the right questions to find the information he or she needs.

(b) To collect data ***from existing documents*** various techniques are used. The most common are the NLP (Natural Language Processing) techniques and techniques of the AI (Artificial Intelligence). Sometimes also a design documentation of software systems is analyzed (manuals, design documentation, software models or meta-models,

repositories etc.). The biggest problem of these approaches is the ambiguity of natural language and therefore there is always a need for additional control and adjustments by a human expert. Finally, it is necessary to gain existing documents or software documentation to be analyzed and they may not even exist for a concrete domain.

(c) Finally also an analysis of *existing software systems* is carried out. The approaches use mainly a static analysis of source codes or databases. Databases or source codes are however *not primarily intended for the user*. Hence the author of database or code is **not forced to use the domain vocabulary** during development. Therefore the domain terms may not even be included in the database or source code, or they can be written in another language, abbreviations can be used, or there may be other language barrier.

The analysis of databases depends on a database schema quality as well as the semantics associated with it. Assuming, that the target database is well-designed with respect to domain, the analysis of such database is easy: the names of tables and relations between tables can be exactly determined and they represent domain terms and relations. There is, however, *no description of domain processes*.

Source codes on the other hand contain descriptions of domain processes in a form of methods or functions. There is however a *high level of implementation details* getting in the way, preventing the domain analysis to be carried out. *Generalization*, which is currently widely used in the implementation of reusable systems, represents another barrier. The aim of generalization is to use generic terms that can be used to describe objects and thus to ensure reusability of the system (or parts of the system) also in other domains. This may hinder the domain analysis: a domain-specific model should contain domain-specific terms, not general.

1.3. Proposed solution

There are many methods and tools supporting the communication with users and domain experts. Starting with manual techniques like interviews and questionnaires, continuing with feature-based methodologies [10, 13], UML modelling [24], CASE systems [20], requirement modelling and analysis [16] and ending with supporting systems like ToolDay [18] or Sherlock [29]. In the area of the document analysis there is a significant number or research papers (many of them were summarized in our previous work [4] in Section *B. Related work*). There is also a possibility of analyzing the system documentation, but due to the trends of rapid software development many programmers nowadays to save time rather develop systems without any documentation. Therefore software documentation may not be available for a particular system. Our decision was to focus on the last area, an analysis of existing software systems. Many approaches dealing with this topic were analyzed in our previous work [4] and other ones will also be summarized in the Section 2. None of them is dealing with user interfaces specifically.
Based on the above facts we argue that a more appropriate target for DA is a user interface of a software system. A *user* who comes from the target domain has a *direct access to the UI*. For the user to be able to use the system effectively it must be built with respect to *understandability*, i.e. it *must contain terms from the domain*. It also should *describe domain processes* in a form of event sequences, which can be executed on the UI. Using of reflection and aspect oriented programming [14] allows us to *separate the implementation details* from domain information needed.

1.4. Motivation

Besides the tasks of creating a new software system, or editing an existing system, the resulting domain model can also be used in other areas, which are also the aim of our future work. The domain model can be used for example in model driven engineering (MDE) [26] for *generating various software artefacts or whole software systems*. For example Ristic et al. [25] use form specifications to generate forms. However when using their tool the user alone must enter the form specifications into the computer. An automatic domain analysis of existing software systems with forms (for example web sites) for such form specifications could help the user with formalizing the requirements and the user would only edit the results of the domain analysis according to his or her needs.
If there already is an existing software system, based on the domain model of such a system and with the help of dynamic analysis of a user handling this system and generators, a *documentation* (for example a user guide) and different types of models usable for future development can be generated.
Generating a whole user interface for another type of application is another form of use. A case scenario: a company develops applications for BlackBerry devices. The owners of the company decide to widen their manufacture for Android

mobile devices and to all applications developed until now should be developed also for the Android platform. A way of making this process easier is to use automatic GUI analyzers and generators to automatically generate user interfaces for Android platform and then manually complete the source code of each generated GUI.

1.5. Organization of the paper

The paper is further organized as follows: Section 2 is a state of the art in the field of domain analysis. In the Section 3 our previous research in the area of automatic GUI formalization is described. Section 4 defines user interfaces and provides a typology of user interfaces. Subsection 4.5 lists common terminology used in the rest of this article. Next Section 5 is devoted to graphical components and their meaning as in the user interface domain-specific units. The section provides a typology of components and describes them in a simple example. The Subsection 5.1 explains the basic principle of identifying stereotypes of creating UIs in a simple example of a Person dialog. At the end there is a list of all identified stereotypes and also a list of corresponding facts and rules that were derived based on each stereotype. The Subsection 5.2 outlines our method for extracting domain-specific information from user interfaces named DEAL and a prototype implementing of this method is presented. The Section 6 provides insight into our future research and Section 7 contains conclusions.

2. State of the art

The domain analysis was first defined by **Neighbors** [22] in 1980 as "the activity of identifying the objects and operations of a class of similar systems in a particular problem domain". Neighbors later introduced the concept of "domain analyst" [23] as the person responsible for conducting domain analysis. By introducing the Draco approach, a code generator system that works by integrating reusable components Neighbors stresses, that domain analysis is the key factor for supporting reusability of analysis and design, not the code.
The most widely used approach for domain analysis is the **FODA** (Feature Oriented Domain Analysis) approach [13]. FODA aims at analysis of software product lines by comparing the different and similar features or functionalities. The method is illustrated by a domain analysis of window management systems and explains what the outputs of domain analysis are but remains vague about the process of obtaining them. Very similar to the FODA approach, and practically based on it, is the **DREAM** (Domain Requirements Asset Manager) approach by Mikeyong et al. [20]. They also perform commonality and variability analysis of product lines, but with the difference of using an analysis of domain requirements, not features or functionalities of systems. Many approaches and tools support the FODA method, for example Ami Eddi [7], CaptainFeature[1], RequiLine[2] or ASADAL[3].
There are also approaches that not only support the process of domain analysis, but also the reusability feature by providing a library of reusable components, frameworks or libraries. Such approaches are for example the early **Prieto-Díaz approach** [8] that uses a set of libraries; or the later **Sherlock** environment by Valerio et al. [29] that uses a library of frameworks.
The latest efforts are in the area of **MDD** (Model Driven Development). The aim of MDD is to shield the complexity of the implementation phase by domain modelling and generative processes. A presumption is that a mapping must be created between domain model and the desired solution. This way they support reusability: in an ideal case, all the user has to do is choosing functionalities he wants to have in the resulting system and the system will be automatically generated. The MDD principle support provides for example the Czarnecki project Feature Plug-in [1, 6] or his newest effort Clafer [5] and a plug-in FeatureIDE [17, 27, 28] from Thüm and Kästner.
ToolDay (A Tool for Domain Analysis) [18] is a tool that aims to support all the phases of the domain analysis process. It has possibilities for validation of every phase and a possibility to generate models and exporting to different formats.

[1] *The webpage of CaptainFeature SourceForge.net project, https://sourceforge.net/projects/captainfeature*
[2] *The webpage of RequiLine project https://www-lufgi3-informatik.rwth-aachen.de/TOOLS/requiline/index.php*
[3] *A review of ASADAL CASE tool, Postech Software Engineering Laboratory, http://selab.postech.ac.kr/ASADAL-Simple_Overview.pdf*

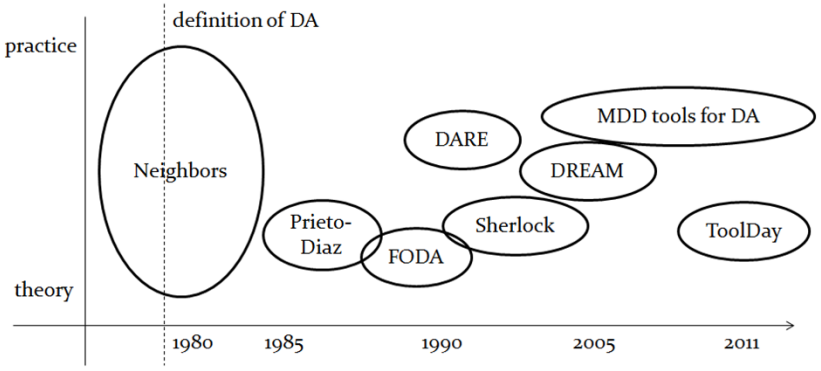

Figure 2. A timeline of existing approaches and tools for domain analysis.

All these tools and methodologies support the domain analysis process by analyzing data, summarizing, clustering of data, or modelling features. But the input data for domain analysis (i.e. the information about the domain) always come from the users, or it is not specified where they actually come from. Only the **DARE** (Domain analysis and reuse environment) tool from Prieto-Díaz [10] primarily aims at automatic collection and structuring of information and creating a reusable library. The data are collected not only from human resources, but also *automatically from existing source codes and text documents*. But as mentioned above, the source codes do not have to contain the domain terms and domain processes. The DARE tool *does not analyze the user interfaces* specifically.
A timeline of the approaches can be seen in Fig. 2.
Very interesting process is also seen in [30] where authors transform ontology axioms into application domain rules which is a reverse process compared to ours.

3. Our previous research

As showed by our previous research [2, 3], the difficulty of GUI analysis can depend on the style of programming or on compliance of programming guidelines. In this work the component approach for formalization of user interfaces of Java applications was practically verified. A tool was created with a support for semi-automatic formalization of UI into a form of a simple domain-specific language GUIIL. More about domain-specific languages can be found in [9, 15, 19]. Identification of components and extraction of information was implemented using reflection. The tool was able to display a simple form of a component tree.
The tool was experimentally verified on a number of open source Java applications (Java scientific calculator, Java notepad, jEdit, jEdit installer, Home3D, GuitarScaleAssistant). Two of them (jEdit and Home3D) were using their own class loaders which prevented the analysis. The problem was resolved by creating external adapters using aspect oriented programming. In GuitarScaleAssistant there was a problem caused by a custom component created by the author of the application – the get and set methods were missing in the component's implementation therefore the analyzer was unable to identify the component's value. The problem was resolved by implementing a specific adapter for this component and using aspect oriented programming.
The result of this tool, i.e. a file written in GUIIL language, was a simple domain model. It contained a list of component identifiers (if these identifiers existed in the target application), which were displayed in the UI. It did not describe a domain structure nor the domain relations, it represented however *simple domain processes in a shape of command sequences*.
The results of our previous research showed that it is possible to identify components in the UI and to record domain-specific information stored into components by their author, but only under certain conditions. These include: access to these information; and correct implementation of components, so they contained the domain-specific information. These problems can be resolved using the aspect oriented approach.

4. User Interface as a source of domain information

A **user interface** (UI) represents the layer of an application designed to interact with users. Based on their form, the UIs can be divided into 4 basic groups described in the next subsections. Only these groups are relevant for our current research.

4.1. Console UIs

It is written "UIs", not "GUIs" because these are not fully "graphical" UIs. They are obsolete text interfaces that use no graphics. A domain analysis of such interface is simple: because there are no graphics involved, the analysis is performed at the source code level. There are approaches doing this type of analysis, for example Moore [21] transforms the console interfaces into WIMP interfaces (more about WIMP UIs in section 4.2), but it is a research from 1997, quite obsolete. The console UIs are rarely used today therefore they are not an important part of our current research.

4.2. WIMP GUIs

The "WIMP" abbreviation stands for "Windows, Icons, Mouse, Pointers" and it represents standard desktop applications with windows. Difficulty of a domain analysis of such interfaces depends on the programming language used and on the programming style of their author. As shown in our previous research in this field ([2, 3]), if the author of UI follows good programming habits and programming guidelines (for example like Java Look & Feel Design Guidelines[4]), the possibility of success of DA increases. And vice versa, it decreases when analyzing custom components designed by a programmer. Using advanced implementation features, such as creating an own class loader, prevents the analysis and restricts it to using only aspect-oriented programming.

4.3. Web GUIs

In the context of this article HTML documents are referred to in two different ways. The first is a *web resource*, that is, a simple web document with no interactive elements other than web links. The second term is a *web application* representing an application with an interactive graphical user interface. Such interfaces will also be called *web-based user interfaces*, or *Web GUIs*. The difference between these two concepts can sometimes be very small or none. Interaction with the Web interface is performed simply by clicking on web links, or (in the case of interactive web applications) similar to WIMP applications – but the content of the UI remains in a web browser (no windows). Interactive web applications are those using advanced technologies such as Flash, JavaScript and so on.

4.4. Mobile GUIs

Mobile devices use a completely different and simpler kind of user interfaces, mainly because of hardware limitations of mobile platforms. There are two types of mobile devices (and hence two types of UIs also) – non-touch screen devices and touch screen devices. Both types of mobile interfaces in most cases consist of *screens* (a Screen class type), which can be displayed on (pushed) or hidden from (popped) the device display. Each component of Screen type has its title and its content. The content is organized using managers and it is a hierarchy of graphical components. A domain analysis of such interfaces is rather simple: the screen and its components clearly form a hierarchy of terms. But the process of connecting analyzer to a mobile device or a mobile device simulator could be complicated. But such analysis would be similar to traditional WIMP UI analysis.

[4] *Java Look & Feel Design Guidelines, version 2.0, http://java.sun.com/products/jlf/ed2/book*

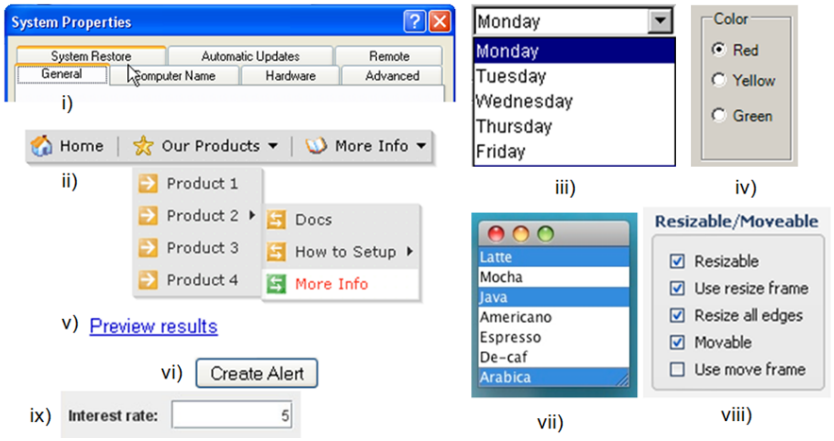

Figure 3. Examples of graphical components: i) a window with a tab pane, ii) a menu and menu items, iii) a combo-box (with one choice selection), iv) radio buttons, v) a web link, vi) a button, vii) a list (with multiple selection), viii) check box buttons (with multiple selection), ix) a text field with a label.

4.5. Terminology

In this article all dialogs, windows, mobile screens or web pages will be described with one term – a **scene**. This term was also used by Kösters in [16]. Each UI can contain one or more scenes and each scene contains one or more components. The components can be of two types – a **component**, which do not contain other components and a **container**, which is a component that can contain other components. A scene is also a container component. By creating different scenes and by inserting components into scenes and containers, a **hierarchy of components** is created.

5. Graphical components as domain-specific units

Components are the fundamentals for component-based software development (CBSD) [11]. In this approach the aim is to develop applications by *assembling components* to a bigger unit while reusing as many components as possible. The main advantages of CBSD are a reduction of price and time, a higher quality and an easier maintainability. A **component** is an independent and functional part of an application that has specific functionalities provided through a clearly defined interface and it has specific dependencies on the surrounding environment, that are defined by the interface. In applications based on components the graphical user interfaces also consist of components. Concretely they consist of a special type of components - **graphical components** - which represent functional parts of GUI (for example buttons, menu items, labels, text fields, etc.). The same as for the classic components holds also for graphic components, i.e. they have a well-defined interface and they are depending on the surrounding code. Each graphical component also contains a definition of its appearance and information, that appears in the interface, i.e. to the user.
There are several types of components (examples can be seen in Fig. 3):

- *informative and text components* – text fields, labels (ix in Fig. 3)
- *functional components* – buttons, web links, menu items (ii, v and vi in Fig. 3),
- *components groupng their content graphically or logically* – for example Windows, dialogs, tab panes, menu, button groups, lists of items (i, ii, iii, iv, vii, viii in Fig. 3),
- *custom components* – components created by a programmer.

A GUI can be imagined as a *tree of graphical components*, where each component represents a *domain-specific unit*. These domain-specific units can be seen as *terms of the GUI language*. For these terms there are specific *rules*, specific *relationships* exist between them and their *hierarchy* is exactly determined. For example a form and a button represent

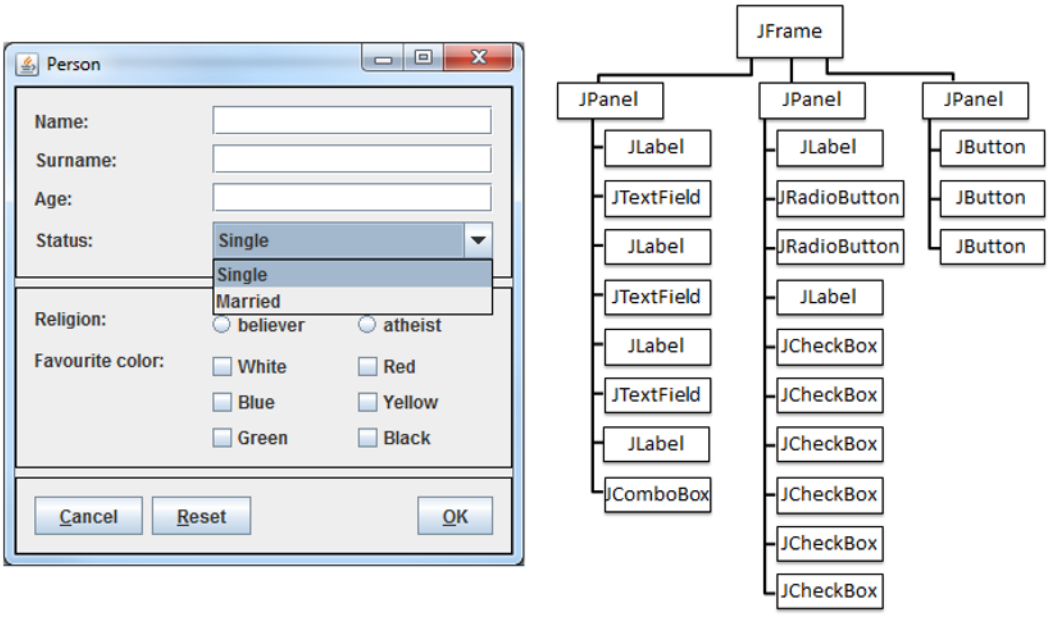

Figure 4. A person form and mapping its components into a component tree.

terms. Their relationship is, that the button belongs to a form. The *hierarchy* is created by including a component into a container (for example a form), which defines the membership of the component in a logically related group. The basis of a hierarchy is a *scene*, which defines a domain or a sub-domain. A root of the UI component tree is the basic scene, which opens when you start the application. Actions performed on this scene can invoke other scenes to open, that are placed this way into the hierarchy of scenes under the basic node. A component tree of a simple person form can be seen in Fig. 4.

5.1. Stereotypes of creating GUIs

This section will summarize the basic stereotypes of creating GUIs, which lead to inserting domain-specific information to the GUI by programmers. It also contains principles, which will enable recording this information during the automated domain analysis.

The first and basic stereotype of creating GUIs is **creating a scene and giving a title to a scene**. Scenes are special types of containers that can have a title, for example a window, a dialog, a web page or a mobile screen. The title defines the domain or sub-domain of the scene. For example a window title defines the window domain or sub-domain. Or a web-page title defines the web-page domain or sub-domain. For example, in the Person dialog in Fig. 4 the name of the dialog defines the domain or sub-domain of all content of the dialog: the dialog contains information about a person.

Next fundamental principle of creating component-based GUIs is **putting all components related to a scene into the scene**. In other words, all relevant information (i.e. only information that is needed to create a scene) related with the scene domain should be placed into the scene. That means all components in the scene represent domain-specific terms that are related to the scene domain.

The components in the scene are often structured, i.e. placed into some form of a container. In Java there are the Container components, in web there are forms or frames, in mobile devices there are managers. Often the **containers create groups of components that are logically or graphically related**. For example, in the person dialog in Fig. 4 there are three containers, the first container contains the basic information about a person, the second container holds some additional information and the third container contains three functional units, related to the dialog.

Based on the presumption that a GUI is made of scenes and that all components are put into the scene and they create

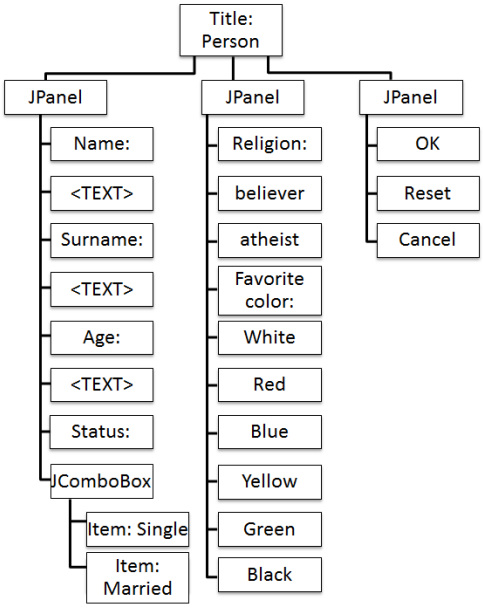

Figure 5. An extended component tree. The terms were derived from components and the additional items were taken from the combo-box component.

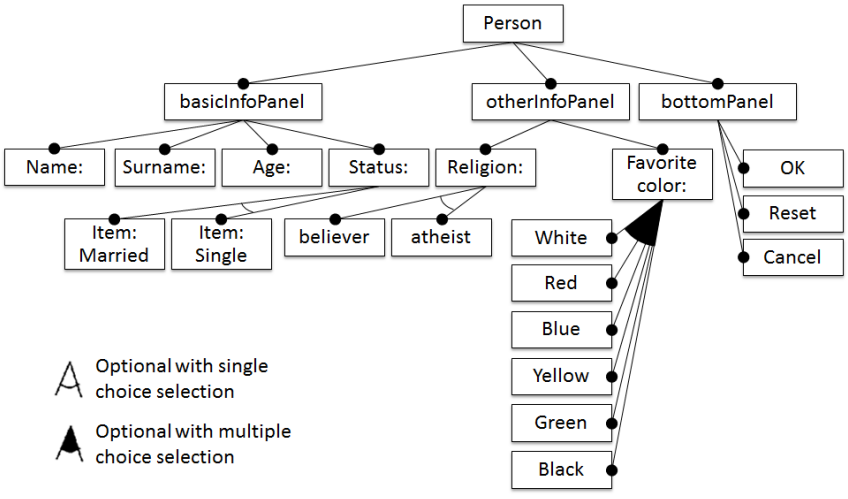

Figure 6. A FODA diagram of the person form.

a hierarchy of containers and components, a **simple component tree** can be derived. However the desired output is much more complex: the domain terms represented by components and also *additional relations* need to be derived. See for example Fig. 5 with a little more complex component tree. All the domain terms were derived from the component's attributes and also new items under the status combo-box were added, *single* and *married*. The combo-box contains these two items and the list of items can be derived from the combo-box component. However this is not the end yet. The combo-box component allows only selection of one item therefore the single and married items are *mutually exclusive terms*. And that's true; in the real world under normal circumstances a person cannot be both single and married at the same time.

Also in the real world, the single or married items represent a relationship status of a person. However the combo-box

does not contain such information. But it can be derived from the label-component relationship. **Labels are commonly used to describe components in forms.** For example the status label describes the combo-box component. That can be derived for example from the label's *labelFor* attribute. If there is no such attribute set, then it is possible to analyze components from the *graphical point of view* and to determine their position in the UI. The status label is in one line with the combo-box component, therefore the status label describes the combo-box component.
As seen in this simple example with combo-box, different kind of rules for extracting domain-specific information can be derived from the type of component and from the way the components are commonly used. First the stereotypes of using these components to implement GUIs must be defined. The rules for extracting domain-specific information will be derived based on these stereotypes. In Table 1 there is a list of stereotypes of creating GUIs in the first column and in the second column there are the rules or facts that were derived based on these stereotypes. These rules will serve for defining the extracting algorithm.
In the table these relations are listed:

- The *belongs-to* (or *parent-child*) relation is a relation between a parent node and a child node in the component tree.
- The *is-related-to* relation represents a relation between two elements, which are somehow linked together logically. For example, they belong to the same domain.
- The *is-a-functionality-of* relation between a scene and a functional component defines that the functional component represents one single functionality of the scene.
- The *mutually-exclusive-to* and *not-mutually-exclusive-to* relations are specified for a list of terms with many options. If only one item can be selected from the list, then all the items are mutually exclusive. If multiple items can be selected from the list, then all the items are not mutually exclusive.
- A *label-for* relationship between a label and a component defines that the label describes the component or provides an additional information about the component.

5.2. DEAL - The Domain Extraction ALgorithm

The extraction process is based on a presumption that the target user interface consists of components and it is window-based and of an object-oriented nature. For web user interfaces another approach using HTML parsers must be used. Also it should be noted that the algorithm is not fully automated, but rather assisted: the user clicks on a running application to invoke opening new windows or dialogs. A prototype of DEAL (Domain Extraction ALgorithm) is already implemented and a simplified description will be presented here. The algorithm is implemented in Java and it uses AspectJ for attaching to an existing domain application. Generics are used for traversing components.
The DEAL prototype contains two aspects: **MainAspect** for attaching the window listener to the target application and **ModelGeneratorAspect** which in case of opening a new window or dialog creates a new domain model and adds it to the viewer. It also contains a GUI for displaying the generated domain model. Each time a new window or a dialog appears, a new node in the domain model is added and displayed in the DEAL application GUI.
A traversal algorithm which is the main process of DEAL is executed for each new opened window or dialog. Algorithm 1 represents the pseudo-code for the traversal algorithm. The traversal algorithm has two cycles: in the first one a search for superclasses is performed to find the appropriate class for which the handler exists. If there is a handler, then a feature is added into the tree. In the second cycle a traversal of all subcomponents is performed. The cycle will be executed only if the component is a composite. For each subcomponent the same procedure will be executed.
After executing the traversal algorithm a domain model in a form of a feature tree is created. A feature is a representation of a component that contains information about its title, label, description, toolTipText and type (class). Each feature can contain other features. The last phase of DEAL is a simplification of the model:

- nodes that represent containers and do not contain any children are removed
- nodes of type container that have only one child of type container are replaced with its child

The DEAL prototype uses handlers for handling different types of components. There are two basic types of handlers:

- Composite – if a component is of type composite (a container), then it can contain other components. The Composite handler gives the list of subcomponents of the given composite. The handler contains only one method **getComponents()**.

Table 1. A list of stereotypes and facts derived from the stereotypes .

Stereotypes	Derived facts
The GUI is made of **scenes**. Scenes are special types of containers that can have a title.	Domain model of an application contains models of domains defined by the scenes (i.e. scene domain models).
The **title** of a scene defines the domain or subdomain of the scene.	A scene domain model has a domain specified by the scene title. Each term in the scene belongs to this domain.
The scene is made of **components**.	The components represent domain-specific terms of the domain model.
The components in the GUI are structured using **containers**. The components placed in containers are often related.	The basic relations between terms are defined by the hierarchy of containers. Two basic types of relations defined by containers: • a *belongs-to* relation: between the container and a component placed in the container, • an *is-related-to* relation: between two or more components placed in the container.
Functional components are used for creating the functional parts of an application. Examples of functional components are buttons, menu items or web links. In domain-specific applications the common words (like "Open", "Close", "Save", "Save as...") are used for describing functional elements, but also domain-specific terms (like "Bibliography", "Furniture", "Gadget"). Functional components placed together in a container are often related to each other (for example menu items in a menu).	The functional components define the basic set of functionalities of the scene. Grouping of these components can be defined by different principles: • menu items are grouped together by menus or submenus, • buttons or web links can be grouped together: – graphically (by putting the components near the others), – by putting them into a scene, – or by a container. The grouping creates an *is-a-functionality-of* relation between a scene and a functional component. If a menu is a context menu, then the relation is between a component, on which can the context menu be displayed; and the context menu items. After removing all commonly used descriptors of functional elements like "Open", "Close", "Save", "Save as...", etc., a *domain functionality* of a scene can be derived.
The **tabbed panes** contain different views and the user is able to switch between the views by clicking on the tab title. The whole tabbed pane is a container and each tab is also a container. Tabs usually have a title. Commonly, the information contained in one tab is related, but it is not related to the information contained in other tabs.	The tab title defines the domain of the content of the specific tab view (similar to a scene and scene title). A tab defines the same relations as the container (*belongs-to* and *is-related-to* relations). A tabbed pane defines a *mutually-exclusive-to* relationship between the domains defined by tab titles and between the content of different tabs.
The **labels** are used to describe components.	If there is a label, that describes a component, than every domain information provided by this component is described (or defined) by the label text. A *label-for* relation can be determined by: • label attributes (for example a *labelFor* attribute in Java or a *for* attribute in HTML), • graphical position.
The **radio button groups** are used to offer options with a single selection. The same holds for **combo-boxes** and **lists** with a single selection or **spinners**.	The label of a radio button (or an item from a combo-box or from a list) represents a domain-specific term. There are two types of relations defined by radio button groups: • an *is-related-to* relation between the terms represented by radio buttons • a *mutually-exclusive-to* relation between the terms represented by radio buttons
The **check-box button groups** are used to offer options with a multiple selection. The same holds for **lists** with multiple selection.	The label of a check-box (or an item from a list) represents a domain-specific term. There are two types of relations defined by radio button groups: • an *is-related-to* relation between the terms represented by check-box buttons • *mutually-not-exclusive* relation between the terms represented by check-box buttons

Stereotypes	Derived facts
Textual components (i.e. text fields, text areas or textual spinners) are used to gain text information from the user.	Following information about the text components can be determined: • their purpose (from their description, if there is any), • restrictions for their content (for example minimal and maximal length, regular expression) from component restrictions, if there are any, • type of content (text, numbers, password), • content, if there is any.
A **tree** is used for displaying structural data, for example a file system or an offer of goods in a shop.	The tree defines an *is-related-to* relationship between all its nodes because they all belong to a same domain. It also defines a *belongs-to* or *parent-child* relationship between the nodes, that is defined by node nesting. If a tree has a description in some form (for example label or a tooltip text), then this description defines a domain of the tree and its nodes.
Forms are used in HTML commonly to gain some information from the user. Usually they contain normal components, like labels, check-boxes, radio buttons, text components, buttons and menus. Forms can be submitted and by submitting they are sent for processing. They also can be reset, i.e. brought to their initial state. A form can have a title. Some information in the form can be set to be required.	A form is similar to a (sub-)scene. The same rules as for the scene hold also for the form. A title of the form defines a (sub-)domain of its content. If a component in the form is required, then it defines a *required* rule for the data described by the component label.
Custom components are implemented to create a functionality that is not supported by default components and that is specific for the domain application.	By implementing custom components the programmer inserts new terms into the GUI language. The domain information defined by these components is specific and it depends on the specific component implementation.

Algorithm 1 Traversal algorithm

Input:
- *component* - a window or a dialog as a root component to traverse;
- *rootFeature* - root node of a feature domain model

Output:
- *domainModel* - a domain model in a form of a root feature

```
componentClass ← class of component;
while componentClass ≠ null do                    ▷ first cycle: traversing superclasses to find a handler
    handler ← Handler for componentClass;
    if handler ≠ null and handler instanceof DomainIdentifiable then
        feature ← createFeature(handler);
        add feature to rootFeature
    end if
    add treeNode for thisComponent;
    componentClass ← superclass of componentClass;
end while

composite ← Composite handler for component;
if composite ≠ null then          ▷ if there is a handler, then it has the getComponents() method
    if feature = null then
        if composite ≠ null and composite instanceof domainIdentifiable then
            feature ← createFeature(composite);
            add feature into rootFeature;
        end if
    end if
    for each subcomponent of composite.getComponents() do      ▷ second cycle: traversing subcomponents
        call procedure Traversal algorithm(subcomponent, feature);   ▷ and recursively calling this procedure
    end for
end if
```

Figure 7. The Java Scientific Calculator used for the experimental verification.

- CommandHandler – handles commands executed on a component (this is related to our previous research). The handler contains the **execute()** method.

For ensuring, that the domain information will be extracted for each component differently a **DomainIdentifiable** interface was implemented. Each Composite or CommandHandler can implement this interface. The CommandHandler contains following elementary methods:

- getDomainIdentifier(T component) – returns the identifier of the component (for example a label of a button).
- getDomainDescriptor(T component) – returns the descriptor of the component (for example a toolTipText of a button).
- getDomainLabelDescriptor(T component) – returns the label descriptor of the component (for example a text of a label of a text component).

Five basic handlers for Java applications are currently supported by the prototype: AbstractButton handler, Container composite, Dialog composite, Frame composite, and JMenu composite. This algorithm was tested on an open-source Java Scientific Calculator (see Fig. 7) which is implemented as an applet. The labels of buttons were combined with HTML tags and this has worsen readability. Therefore also three domain-specific handlers for the Calculator application were created to be able to properly extract domain-specific information: CalculatorButton handler, CalculatorDisplay handler and CalculatorEntryLabel handler. The created model is in a form of a tree of domain terms representing components. The result can be seen in Fig. 8. The additional information about components (for example toolTipText or a type) is displayed as toolTipText on each feature, see Fig. 9. The DEAL prototype is only a starting point for implementing the domain analyzer but the results show that the proposed solution is feasible.

6. Further research

In our previous and current work mentioned above an algorithm was created for extracting domain terms from each type of component. A domain term can be extracted from different attributes: *name*, *title*, *toolTipText*, *label* or *actionCommand*. Priorities of the attributes were defined and an algorithm was created which consisted of a number of sub-processes for information extraction. The tool created in our previous research was able to display a basic hierarchy of terms in a form of a component tree. There was only the *belongs-to* relation between the tree nodes derived from the hierarchy of components and containers in a scene. A model of a component tree of the person form is displayed in Figure 1.
In this article we described the most common stereotypes that often occur when creating GUIs. Based on them we derived a series of rules or facts describing the possibilities of extracting domain information from GUIs. We presented the pseudocode of our DEAL algorithm and a DEAL prototype, which is able to extract a tree of components in a form of a feature diagram. The prototype supports five basic Java components and three additional handlers for Java Scientific Calculator were created in order to be able to analyze it.

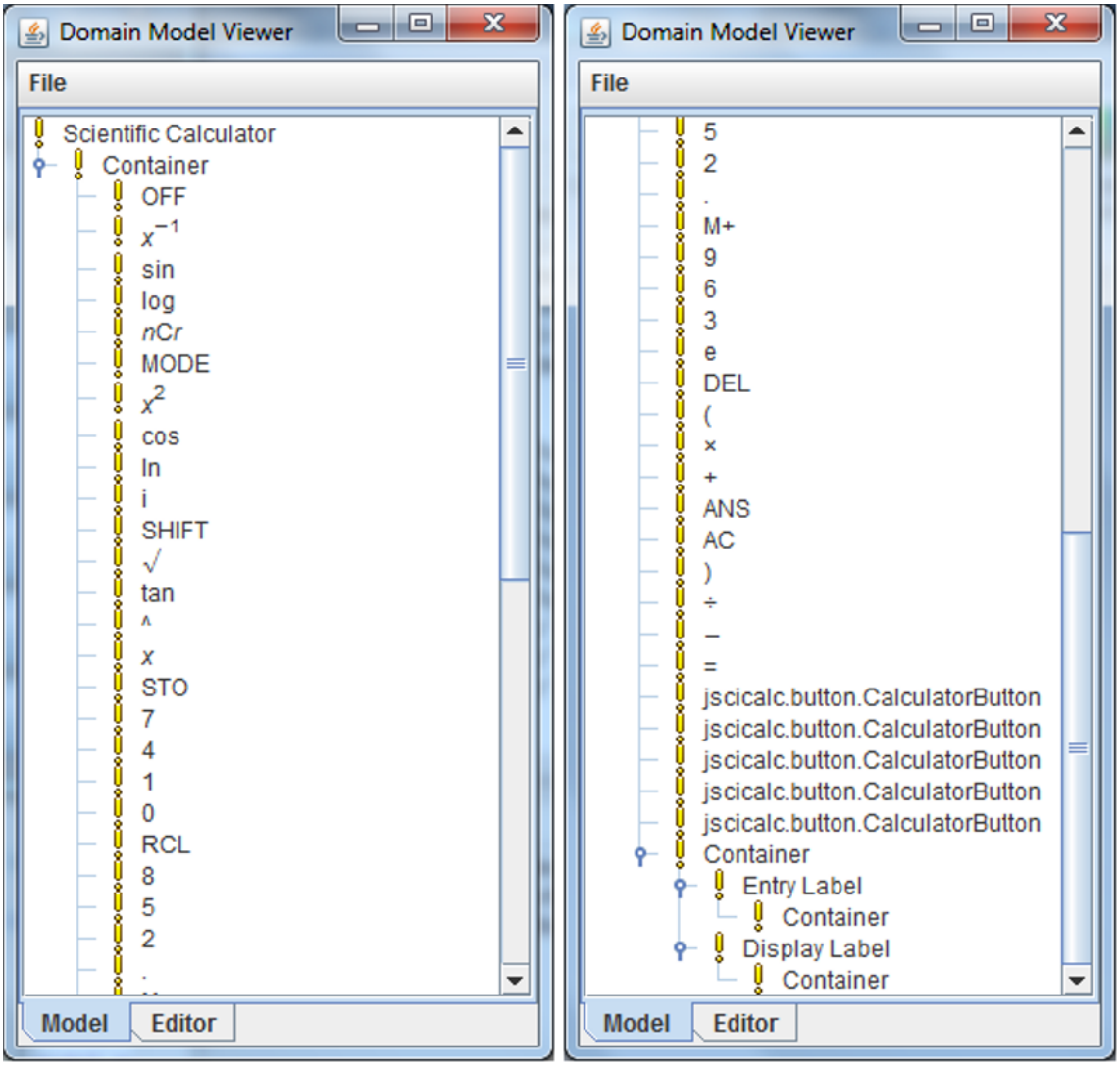

Figure 8. A result of an analysis of the Java Scientific Calculator from fig 7 displayed as a feature model in a Feature plug-in notation.

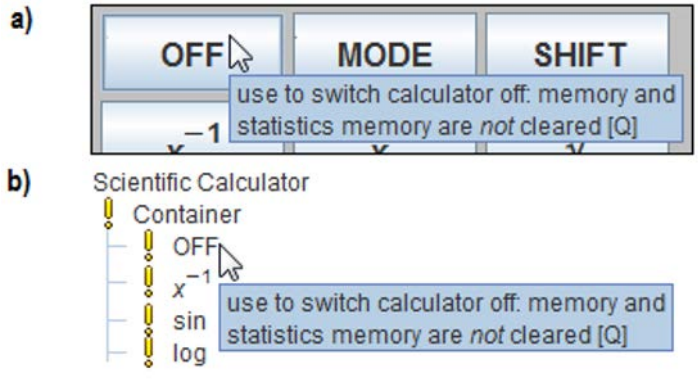

Figure 9. Additional information about the domain term is displayed as a tooltip. Picture a) is taken from the Java Scientific Calculator and picture b) is taken from the DEAL display.

The next step is to implement new handlers for different components to be able to support wider functionality. Also for components that define relations between terms (like radio buttons, check-boxes, tab panes) an additional functionality of the handler must be implemented, that defines these relations to be able to extract it into the feature diagram. Further we will also continue to expand the number of stereotypes to make the DEAL algorithm more flexible. In Fig. 6 there is an example of our research for future actions. There is a domain model (in FODA form - FeatureIDE notation) of the person form derived from the component tree in Fig. 4 and based on the rules listed in the Table 1. The ¡TEXT¿ represents a real text value or the type of a value of the given term. In the end we would like our tool to support different kinds of user interfaces starting with web.

7. Conclusion

In this paper we identified the most common problems in the domain analysis process during extraction of information from different sources. We conducted an analysis of the state of the art in the area of automated domain analysis methods and tools and we concluded that the least explored area is related with existing software systems. We proposed a DEAL method for analyzing user interfaces which represent a very good source of domain information because of their purpose – to serve the domain users. We defined the general goals of our research and also specific goals of this paper and we also explained our previous research in this area. We provided a theory of graphical user interfaces representing a source of domain information using a component approach. At the end we identified the most common stereotypes of creating GUIs and based on them we derived facts and rules, that help us design the DEAL method. A raw description of the DEAL method and of the the prototype was presented here. We also provided an insight to our future research.

Acknowledgement

This work was supported by VEGA Grant No. 1/0305/11 - Co-evolution of the artifacts written in domain-specific languages driven by language evolution.

References

[1] Antkiewicz M., Czarnecki K., FeaturePlugin: feature modeling plug-in for Eclipse, In: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange (ACM, New York, NY, USA, 2004)

[2] Bačíková M., Porubän J., Defining computer languages via user interfaces (Technical University of Košice, Faculty of Electrotechnical Engineering and Informatics, 2010)

[3] Bačíková M., Porubän J., Automating User Actions on GUI: Defining a GUI Domain-Specific Language, In: Proceedings of International Scientific conference on Computer Science and Engineering, 60-67, 2010

[4] Bačíková M., Porubän J., Lakatoš D., Introduction to Domain Analysis of Web User Interfaces, In: Proceedings of the Eleventh International Conference on Informatics, INFORMATICS'2011, 115-120, 2011

[5] Bak K., Czarnecki K., Wasowski A., Feature and meta-models in Clafer: mixed, specialized, and coupled, In: Proceedings of the Third international conference on Software language engineering (Springer-Verlag, Berlin, 2011)

[6] Czarnecki K., Antkiewicz M., Kim, Ch.H.P., Lau S., Pietroszek K., fmp and fmp2rsm: eclipse plug-ins for modeling features using model templates, In: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (ACM, New York, NY, USA, 2005)

[7] Czarnecki K., Bednasch T., Unger P., Eisenecker U.W., Generative Programming for Embedded Software: An Industrial Experience Report, In: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering (Springer-Verlag, London, UK, 2002)

[8] Díaz Rubén P., Reuse Library Process Model. Final Report (Electronic Systems Division, Air Force Command, USAF, Hanscomb AFB, MA, USA, 1991)

[9] Fowler M., Domain-Specific Languages (Addison-Wesley Signature Series (Fowler)) (Addison-Wesley Professional, 2010)
[10] Frakes W., Prieto-Diaz R., Fox Ch., DARE: Domain analysis and reuse environment, Ann. Softw. Eng., 5, 125-141, 1998
[11] Heineman G.T., Councill W.T., Component-based software engineering: putting the pieces together (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001)
[12] Jeff Gray J. et al, Domain-Specific Modeling, In: CRC Handbook of Dynamic System Modeling (CRC Press, 2007)
[13] Kang K.C., Cohen S.G., Hess J.A., Novak W.E., Peterson A.S., Feature-Oriented Domain Analysis (FODA) Feasibility Study (Carnegie-Mellon University Software Engineering Institute, 1990)
[14] Kiczales G. et al., Aspect-Oriented Programming, 220-242 (Springer-Verlag, 1997)
[15] Kosar, Tomaž et al., Comparing General-Purpose and Domain-Specific Languages: An Empirical Study, Comput. Sci. Inf. Syst., 2, 247-264, 2010
[16] Kösters G., Six H.W., Voss J., Combined Analysis of User Interface and Domain Requirements, In: Proceedings of the 2nd International Conference on Requirements Engineering (ICRE '96), 1996
[17] Leich T., Apel S., Marnitz L., Saake G., Tool support for feature-oriented software development: featureIDE: an Eclipse-based approach, In: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange, (ACM, New York, NY, USA, 2005)
[18] Lisboa L., Garcia V., de Almeida E., Meira S., ToolDAy: a tool for domain analysis, Int. J. Softw. Tool. Tec. Trans., 13, 337-353 2011
[19] Mernik M., Heering J., Sloane A.M., When and how to develop domain-specific languages, ACM Comput. Surv., 37, 316-344, 2005
[20] Moon M., Yeom K., Seok Chae H., An Approach to Developing Domain Requirements as a Core Asset Based on Commonality and Variability Analysis in a Product Line, IEEE Trans. Softw. Eng., 31, 551-569, 19, 2005
[21] Moore M.M., Rugaber S., Domain Analysis for Transformational Reuse, In: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97) (Washington, DC, USA, 1997)
[22] Neighbors J.M., Software construction using components (University of California, Irvine, USA, 1980)
[23] Neighbors J.M., The Draco approach to constructing software from reusable components, In: Readings in artificial intelligence and software engineering (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986)
[24] Reinhartz-Berger I., Sturm A., Selected Readings on Database Technologies And Applications, In: Information Science Reference - Imprint of: IGI Publishing (Hershey, PA, USA, 2008)
[25] Ristic S., Aleksic S., Lukovic I., Banovic J., Form-Driven Application Generation: A Case Study, In: Proceedings of the Eleventh International Conference on Informatics, INFORMATICS'2011, 115-120, 2011
[26] Schmidt C.D., Guest Editor's Introduction: Model-Driven Engineering, Computer, 39, 25-31, 2006
[27] Thum T., Batory D., Kastner Ch., Reasoning about edits to feature models, In: Proceedings of the 31st International Conference on Software Engineering (IEEE Computer Society, Washington, DC, USA, 2009)
[28] Thum T., Kastner Ch., Erdweg S., Siegmund N., Abstract Features in Feature Modeling In: Software Product Line Conference (SPLC), 2011 15th International (IEEE, 2011)
[29] Valerio A., Succi G., Fenaroli M., Domain analysis and framework-based software development, SIGAPP Appl. Comput. Rev., 5, 4-15, 1997
[30] Vasilecas O., Kalibatiene D., Guizzardi G., Towards a Formal Method for the Transformation of Ontology Axioms to Application Domain Rules, Inf. Technol. Contro., 38, 271-282, 2009