Refactoring the Documentation of Software Product Lines*

Konstantin Romanovsky[1], Dmitry Koznov[1], Leonid Minchin[1]

[1] Saint-Petersburg State University,
Universitetsky pr. 28, Peterhof, Saint-Petersburg, 198504 Russia
{kromanovsky, dkoznov, len-min}@yandex.ru

Abstract. One of the most vital techniques in the context of software product line (SPL) evolution is refactoring – extracting and refining reusable assets and improving SPL architecture in such a way that the behavior of existing products remains unchanged. We extend the idea of SPL refactoring to technical documentation because reuse techniques could effectively be applied to this area and reusable assets evolve and should be maintained. Various XML-based technologies for documentation development are widely spread today, and XML-specifications appear to be a good field for formal transformations. We base our research on the DocLine technology; the main goal of which is to introduce adaptive reuse into documentation development. We define a model of refactoring-based documentation development process, a set of refactoring operations, and describe their implementation in the DocLine toolset. Also, we present an experiment in which we applied the proposed approach to the documentation of a telecommunication systems SPL.

Keywords: Software Product Line, Refactoring, Documentation

1 Introduction

Technical documentation is an important part of commercial software. The Development and maintenance of requirements and design specifications, user manuals, tutorials, etc. is a labor-intensive part of any software development process. Exactly like software, documentation could be volatile and multi-versioned, and it may also have a complex structure. Moreover, documentation is often developed in several natural languages and in different target formats, like HTML, PDF, and HTML Help. The Documentation of a software product line [1] (SPL), which is a set of software applications sharing a common set of features, – appears to be even more complicated than the documentation for stand-alone applications, since it contains multiple repetitions that should be explicitly managed to reduce documentation development effort.

In [2] we presented DocLine – a technology for developing SPL documentation, which supports planned adaptive reuse. By providing an XML-language for

* This research is partially supported by RFFI (grant 08-07-08066-з).

documentation development, DocLine allows a three-level representation of documentation, namely as diagrams of reuse structure, as XML-specification and as generated target documents in PDF, HTML or other format. DocLine also provides process guidelines and is supported by the Eclipse-based toolset.

SPL development is a complicated evolutional process: While new products are developed, existing ones need maintenance and enhancement. The refactoring of SPL architecture and common assets is a popular approach to improving SPLs [3, 4, 5, 6, 7].

The purpose of this paper is to extend the idea of SPL refactoring to documentation development. Indeed, XML-based approaches to documentation development (like DocBook [8], DITA [9]) are becoming more and more popular, while turning documentation into a kind of formal specification. If we extract and explicitly mark up reusable text fragments to be used in newly created documents, the target representation of existing documents should not change (though XML-specification would be changed). Therefore, we consider it reasonable to use the term *refactoring* for such transformations.

In this paper, we propose a refactoring-based documentation development process model, as well as offering a set of refactoring operations. Also, we describe their implementation in the context of DocLine toolset, and discuss an experiment in which we applied the proposed approach to the documentation of a telecommunication systems SPL.

2 Background

2.1 Software Product Line Development

Back in 1976, Parnas noted that it was efficient to create whole product families [10] instead of creating stand-alone systems. In present, this idea is actively being developed. In [11], a software product line is defined as “a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way”.

Product line evolution is an important issue in product line development. At first, top-down methods of product line development were created, for example DSSA [12]. Top-down methods involved starting development with an in-depth domain analysis, identifying potential reuse areas, and developing common assets, and, afterwards, moving to product line members’ development. Such methods require a lot of investment, but ensure flexible reuse, efficient maintenance and new products creation [13], thus bringing significant economic (and time-to-market) gain after the development of several products. Ultimately, new product development could be done just by selecting and configuring common assets [14]. These methods, however, involve serious risks, because, should the number of developed products remain small, investment will not pay off.

Light-weight bottom-up methods were designed to mitigate these risks. They suggest starting from developing a single product and moving to developing a product

line only when product perspectives become clear [15]. To facilitate this move, common assets are extracted from "donors" (stand-alone products) and form the basis of further development. This approach significantly reduces cost and time to market for the first product, but brings in less profit when a number of products increases.

The protagonists of both approaches agree that the need for change in SPL architecture or common assets arises regularly during product line evolution, but this change should not affect the behavior of products. Moreover, in bottom-up approaches such changes are the foundation of the development process.

2.2 Refactoring

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure [16]. It became popular in agile software development methods because it provides an alternative to expensive preliminary design by allowing for constant improvement of software architecture while preserving the behavior of software.

Refactoring helps to fulfill particular tasks, like code structure and code representation refinement (dead code elimination, conditional expressions simplification, method extraction, etc.), and OO-hierarchy refinement (moving a field between classes, class extracting, pulling a field up/down the hierarchy, etc.). Also, there are so called "big refactorings", for example, transition from procedural to object-oriented design.

Refactoring can be done manually, but ensuring that the behavior of a system remains unchanged is not an easy task. As trivial an operation as it may seem like, renaming a method involves finding and modifying all of its calls (including calls via objects of all derived classes) and could affect the entire application source code. There are tools that facilitate refactoring by automating typical refactoring operations while ensuring correctness of source code transformations (correctness here means that the code remains compilable and its external behavior remains the same). Moreover, some toolsets support manual refactoring, including big refactoring, by providing means for automated regression testing (including, but not limited to unit testing).

2.3 Refactoring in Product Line Development

A lot of research focuses on SPL refactoring. In [4], a feature model refactoring method is proposed as a way to improve the set of all possible product line configurations, that is, to maximize the potential for new products creation. In [3], the authors introduce a method of decomposing an application to a set of features for using them in product line development. What is offered in [6] is a set of metrics and a tool for refactoring SPL architecture. All the methods above are aimed to get better product line variability by extracting common assets and improving their configurability.

2.4 XML-Based Approaches to Technical Documentation Development

Many documentation development approaches employ the concept of content and formatting separation, which means that meaningful document constructions, for example parts, chapters, sections, tables, are separated from their formatting (for example, information about a title font and size selection is not part of content; therefore it is defined separately). This idea was introduced well in advance of XML and it was implemented, for example, in the TeX typesetting system [17] by Donald Knuth. Modern XML-based approaches to documentation development make use of this idea as well as supporting single sourcing, that is developing several documents on the basis of a single source representation [18]. The appropriateness of using such technologies is widely discussed by technical writers [19]. Research data in this area show that the advantages of using XML technology in middle-size and large companies justify the cost of their adoption (see, for instance, [20]). In practice, such XML-based technologies as DocBook [8] and DITA [9], are actively adopted by the industry in dozens of large companies and projects, including IBM, PostgreSQL, Adobe, Sun, Cray Inc. [21], Unix-systems distributives, window environments (GNOME, KDE) [22].

3 The DocLine Approach

3.1 Basic Ideas

The DocLine approach [2] was designed for developing and maintaining SPL technical documentation. One distinct characteristic of such documentation is that there are a lot of text reuse opportunities both in single product documentation and among similar documents for different products. DocLine introduces planned adaptive reuse of documentation fragments. Reuse is planned with the help of a visual modeling technique which allows creating, navigating and modifying a scheme of reusable fragments (common assets) and their relations. Adaptive reuse means that common assets can be configured independently for each usage context.

DocLine features an XML-based language DRL (Documentation Reuse Language), intended for designing and implementing reusable documentation. It also offers a model of documentation development process, and a toolset integrated into Eclipse IDE. DocLine was presented in detail in [2]; therefore we now focus on the basic features of DRL which are critical to explaining the refactoring operations we propose. Also we describe a process model for developing product line documentation, provided by DocLine.

In order to implement text markup we use the well known DocBook [8] approach that is a standard de factum in the Linux/Unix world. In fact, DRL extends DocBook to include an adaptive reuse mechanism. DRL-specifications are first translated by the DocLine toolset into plain DocBook format; then, the DocBook utilities are used to produce target documents in a variety of formats (PDF, HTML, etc).

3.2 DRL Overview

The most important kind of common assets in DRL is an *information element,* which is defined as a context-independent reusable text fragment. In order to put a set of information elements together, DocLine provides what is called an *information product,* which is a template of a real document, like a user guide or a reference manual. Every information element could be included in any other information element or information product. These inclusions can be optional or mutually dependent, and all such variation points must be resolved to derive the specific document from an information product.

DRL provides two mechanisms of adaptive reuse: customizable information elements and multi-view item catalogs.

Customizable information elements. Let us look at the documentation of a phones product line with a CallerID function. It could have an information element named "Receiving incoming calls" containing a text like this:

Once you receive an incoming call, the phone gets CallerID information and displays it on the screen. (1)

Phone may have additional ways of indicating a caller, e.g. a phone for the visually impaired could have a voice announcement instead of a visual presentation, and thus the example (1) would look as follows:

Once you receive an incoming call, the phone gets CallerID information and reads it out. (2)

To facilitate the transformation of the sentence in (1) into the sentence in (2) using an adaptive reuse technique, the corresponding information element must be written in the following way (in the syntax of DRL):

```
<infelement id=CallerIdent>
  Once you receive an incoming call, the phone gets CallerID information
  <nest id=DisplayOptions> and displays it on the screen</nest>.
</infelement>
```
(3)

In this example, we define an information element (<infelement/> tag) and an extension point inside it (<nest/> tag). When this information element is included in a particular context, any extension point can be removed, replaced or appended with custom content without having to modify the information element itself. If no customization is defined, the information element in (3) will produce the text as in (1) seen. The following customization transforms it into the text in (2):

```
<infelemref  infelemid=CallerIdent>
  <replace-nest nestid=DisplayOptions> and reads it out</replace-nest>
</infelemref>
```
(4)

The example (4) shows a reference to the information element (<infelemref/>) defined in the example (3) and the replacement of the extension point defined in this information element by new content (<replace-nest/>).

Multi-view item catalogs. In the documentation of most software products one can find descriptions of typical items of the same kind, for example, GUI commands. In different documents and contexts they are accompanied by a different set of details. In toolbar documentation, for example, commands are defined as an icon with a name and a description. In menu documentation you will see the name of a command, its description and accelerator key sequence; an online help also will contain a relevant tooltip text that shows up when a user drags the mouse cursor over the command button. All these fragments have common attributes of corresponding items. To allow the reuse of such attributes, DocLine introduces the concept of a *catalog*. A catalog contains a collection of items represented by a set of attributes, e.g. the GUI commands catalog may include a collection of GUI commands, each of them having a name, an icon, a description, an accelerator, a tooltip text, a list of side effects, usage rules, etc. Here is an example of such a collection of GUI commands for some software product represented as a catalog in DRL:

```
<directory name="GUICommands">
 <entry name="Print">
   <attr name="name">Print</attr>
   <attr name="icon">Print.bmp</attr>
   <attr name="descr">This command …</attr>
 </entry>
<entry name=" Save">
   <attr name="name">Save</attr>
   <attr name="icon">Save.bmp</attr>
   <attr name="descr">This command …</attr>
 </entry>
</directory>
```

In addition to a collection of items, a catalog contains a set of representation templates that define how to combine item attributes to get a particular item representation. The template in the example below represents a short notation of GUI commands including only the icon and the name:

```
<dirtemplate name="toolbar_short" directory="GUICommands">
   <fig>Images/<attrref name='icon'/></fig><attrref name="name"/>
</dirtemplate>
```

A representation template contains some text and references to item attributes (<attrref/>). When a technical writer includes a catalog item into a particular context, he or she must indicate the corresponding representation template and the item identifier. Then, the content of the template will be inserted into the target context and all the references to attributes will be replaced by corresponding attribute values.

3.3 Documentation Development Process Model

As we discussed above, there are two major models of product line evolution: top-down and bottom-up. Consequently, there are two models of documentation development. DocLine supports both models, but focuses mainly on the latter since it is more often used in practice. The scheme of the bottom-up model is shown in Fig. 1.

In case of the bottom-up model, a technical writer starts from the documentation of a particular product and does not pay attention to reuse techniques. Priority is given to achieving a specific business goal (e.g., a good documentation package for a concrete product), while further perspectives might not be clear. However, once a need for new products documentation arises, a technical writer can benefit from a reuse technique by analyzing reuse options, extracting common assets, and proceeding with the development of documentation on the basis of the common assets. This is an appropriate moment to adopt the DocLine technique. It is a straightforward task if the documentation has been developed by means of DocBook, or it can be easily converted to DocBook. In other cases the existing documentation should be manually ported to DocBook and then marked up with DRL constructions.

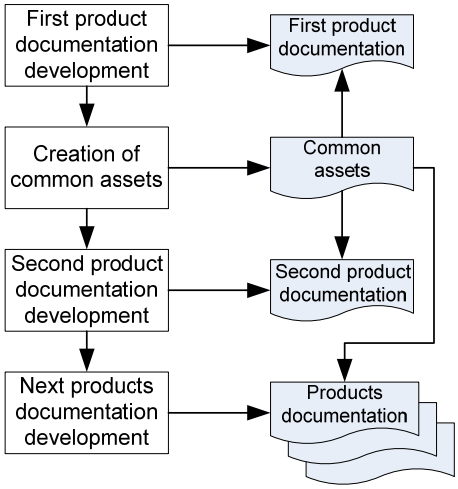

Fig. 1. Bottom-up documentation development process.

4 Refactoring of Product Lines Documentation

4.1 Refactoring Process

Let us look at the bottom-up process model for product line documentation development in greater detail from refactoring perspective:

1. DocLine is adopted when new product documentation is to be created.

2. To use DocLine, a technical writer analyzes the functionality of a new product, and finds similarities, additions and modifications compared to the functionality of existing products.
3. Then, a technical writer finds all fragments in the existing documentation that should be preserved, modified, added or deleted.
4. Then, some formal transformations are applied to documentation sources in order to turn them into correct DRL-specification (if documentation was developed using plain DocBook; otherwise documentation must be ported to DocBook before this step) and to explicitly mark common assets.

Existing and newly created pieces of text which are not reused (or are identically the same) should be converted into a single large information element. DRL-specification is decomposed in-depth if the information element varies from one product to another, or if there are other compelling reasons for decomposition, e.g., simultaneous documentation editing by several technical writers.

The same model remains relevant when a new product is added to a product line, whose documentation has been already developed by using DocLine. In this case, some common assets could be reused "as is" in new product documentation, but typically there is always a need for the refinement of existing assets and the creation of new common assets.

4.2 Refactoring Operations

Let us discuss how exactly refactoring ideas are realized in text transformations. The first group consists of text transformations aimed at extracting common assets.

1. *Extracting information element.* A fragment of text is extracted to a stand-alone information element. Then, it is replaced by a reference to the newly created information element. If the extracted fragment contains extension points, new adapters are created for all final information products to ensure that all manipulations with extension points are preserved.
2. *Splitting information element.* An information element is split into two information elements. All references and adapters are updated.
3. *Importing of DocBook documentation.* The entire DocBook documentation for a product is imported into the DocLine toolset. All the necessary elements are created: the entire documentation is placed into an information element that is included into a newly created information product. Also, a final information product is created. As a result, we receive the same target documentation as the documentation derived from initial DocBook documentation.
4. *Extracting information product.* A selected document (or a fragment) is extracted to an information product. It is followed by the creation of a final information product that uses the newly created element to produce the exact copy of the original document. This operation is normally used when plain documentation in DocBook is imported into the DocLine toolset and a technical writer needs to extract several information products from it.

The following operations are designed to facilitate core assets tuning (extending their configurability).

5. *Converting to extension point.* A text fragment inside an information element is surrounded by an extension point.
6. *Extracting to insert after / insert before.* A trailing / heading text fragment inside an extension point is extracted and inserted into all existing references to the original information element as an Insert After / Insert Before construction. If the text is outside any extension point, but inside an information element, a new empty extension point is created before / after the fragment.
7. *Making a reference to an information element optional. A* particular mandatory reference to an information element (infelemref) is re-declared as optional. In all existing final information products containing the above-mentioned information element, an adapter is created (or updated) to force the inclusion of the optional information element.
8. *Converting to conditional.* A selected text fragment is marked as conditional text (condition shall be provided by the technical writer). In all existing final information products containing the fragment, the condition is set to true to force the inclusion of the conditional text.
9. *Switching default behavior on optional references to information element.* Depending on a technical writer's preferences optional references may be treated as included by default or excluded by default. When default behavior is switched, all adapters must be updated to ensure that the target documents stay unchanged.

The following operations are designed to facilitate the use of small-grained reuse constructions – dictionaries and multi-view item catalogs (directories).

10. *Extracting to dictionary.* A selected text fragment (typically, it is a single word or a word combination) is extracted to a dictionary and its entry is replaced by a reference to the newly created dictionary item. Then, documentation is scanned for other entries of the same item. Found entries could be replaced by references to the dictionary item.
11. *Extracting to multi-view item catalog (directory).* A new multi-view item catalog item is created based on a selected text fragment. Then, the fragment is replaced by a reference to the newly created dictionary item with a selected (or newly created) item representation template.
12. *Copying/moving dictionary/directory item to/from product documentation.* A selected dictionary (or directory) item is copied (or moved) from the common assets to the context of particular product(s) documentation and vice versa. Thus the scope of the item is extended or narrowed.

The following operations facilitate renaming various structural elements of documentation.

13. *Renaming.* The following documentation items can be renamed: an information element, an information product, a dictionary, a directory, a dictionary item, a

directory item, an extension point, a reference to an information element, a dictionary representation template. All references to renamed elements are updated.

4.3 Refactoring Operation Example

Let us consider an example of applying the refactoring operation of extracting information element. The purpose of this example is to show that refactoring operations require non-local source text modifications in order to keep target documents unchanged. Here is a fragment of a documentation source in DRL (note that all these constructions may be stored in physically different files):

```
<InfProduct id="phone_manual">
  <InfElemRef id="basic_ref" infelemid="basic_function"/>
</InfProduct>

<InfElement id="basic_functions">
  <Nest id="connect_options">You can connect your phone to </Nest>
  <Nest id="dial_options">You can dial numbers using </Nest>
</InfElement>

<FinalInfProduct name="office_phone"  infproductid = "phone_manual">
  <Adapter infelemrefid = "basic_ref">
    <Insert-After nestid = "connect_options">
      urban telephone network or office exchange.
    </Insert-After>
    <Insert-After nestid = "dial_options">
      built-in numeric key-pad or phone memory.
    </Insert-After>
  </Adapter>
</FinalInfProduct>
```

This fragment contains an information product *phone_manual* that is the template of the user manual for a phone set. It contains a reference to the information element *basic_functions* describing basic phone functionality. Also, there is a specialization of the user manual: it is a final information product *office_phone*, that uses the *phone_manual* information product to produce a manual for an office phone.

If we run the operation of extracting information element for an extension point (nest) *dial_options*, we get the following changes. The information element *basic_functions* will look as follows (changed text is typed in boldface):

```
<InfElement id="basic_functions">
  <Nest id="connect_options">You can connect your phone to </Nest>
  <InfElemRef id="dial_number_ref" infelemid="dial_number"/>
</InfElement>
```

A new information element is created out of the extracted text fragment:

```
<InfElement id="dial_number">
   <Nest id="dial_options">You can dial numbers using </Nest>
</InfElement>
```

Finally, let us look at changes in the final information product *office_phone* (a changed text is typed in boldface):

```
<FinalInfProduct name="office_phone"  infproductid = "phone_manual">
   <Adapter infelemrefid = "basic_ref">
      <Insert-After nestid = "connect_options">
         urban telephone network or office exchange.
      </Insert-After>
   </Adapter>
   <Adapter infelemrefid = "dial_number_ref">
      <Insert-After nestid = "dial_options">
         built-in numeric key-pad or phone memory.
      </Insert-After>
   </Adapter>
</FinalInfProduct>
```

As you can see, the manipulations with the extracted extension point were moved from the existing adapter to a newly created one that defines adaptations of the new information element.

4.4 The Toolset

Most of the proposed refactoring operations are implemented as part of the DocLine toolset. DocLine toolset is designed as a set of plug-ins for Eclipse IDE [2]. The refactoring tool is embedded into the DRL text editor. In addition to the operations, the refactoring tool provides a framework for implementing new refactoring operations, and a support library that perform tasks typical of most refactoring operations, like DRL parsing (supporting multi-file documentation structure) and DRL generation.

5 The Experiment

The approach to refactoring of SPL documentation presented in this paper was applied to the documentation of a telecommunication systems product line. This product line includes phone exchanges of various purposes: private branch exchanges, inter-city gateways, transit exchanges, etc. For our experiment we selected two product line members – an exchange for public switched telephone network (hereinafter called PSTNX) and a special-purpose exchange (hereinafter called SPX). We decided to port user manuals of these products to DocLine.

During the analysis, we found that historically SPX was developed as a version of PSTNX with reduced functionality. In course of evolution, some functions of SPX were changed, so its user manual was updated accordingly.

First, we converted the documentation to DocBook (in our experiment it was done manually, although for some cases it could have been automated). Then, we started to introduce a DRL markup. We discovered a series of common terms and word combinations in the documentation and created dictionaries and directories to guarantee their unified use across the documents. After that, we "mined" some text fragments, which were similar in both documents, and wrapped them with information element constructions to make them available for reuse in various contexts. Then, we "fine-tuned" these information elements to prepare them for use in both documents.

We used the following refactoring operations in our experiment: importing of DocBook documentation, extracting information element, converting to extension point, extracting to insert after / insert before, making reference to information element optional, extracting to dictionary, extracting to directory. These operations helped us to build an efficient internal structure of the documentation and enable the reuse of text fragments across the two documents while preserving the view of the target documents.

One of our findings is that joining two products to form a family significantly differs from deriving a new product from an existing one to form a family. Let us consider an extracting information element operation. How do we find what to extract? Likely candidates are similar but not identical text fragments, but finding them in two documents with a total of 300 pages proved to be a very difficult task. This suggests that there is a need for a specific tool which a technical writer could use alongside with the refactoring tool, to facilitate finding potential common assets.

6 Conclusions and Further Work

The product line documentation refactoring approach proposed in this paper is designed to facilitate moving from monolithic documentation for one or several products to reusable documentation of a software product line with explicitly defined common assets. It can also be used for developing documentation for newly created product line members.

In our further research we plan to enable intelligent selection of candidates for refactoring: fragments to be extracted as information elements, frequently used words to be extracted to dictionaries and common word combinations to form multi-view item catalogs. What seems to be a promising approach to find candidates for an information element extraction is source code clones detection, since it could be enhanced so as to identify "polymorphic" clones. Techniques for product line variability management are also of interest to us because they could provide a technical writer with information on products variability that is more or less reflected in the documentation structure (e.g. common features in a product variability model correspond to information elements in a documentation).

We also plan to introduce means for “big refactorings” (major changes of documentation composed of series of automated and manual transformations). Entire automation for such an operation is impossible, but we could offer some useful services, for example, the automated checking of target documentation consistency.

Another area for further research is the pragmatics of refactoring, and we would like to propose some ideas of how to guide documentation refactoring. As for program code refactoring, there are coding conventions, rules of building OO hierarchy, off-the-shelf recommendations on using various refactoring operations, etc. We plan to develop a set of similar recommendations for our case. One more question of pragmatics is how to keep a balance between the configurability of documentation and the complexity of its structure.

Finally, we intend making a larger experiment which main goal will be to test the scalability of the proposed approach.

References

1. Northrop, L., Clements, P.: A Framework for Software Product Line Practice, Version 5.0. http://www.sei.cmu.edu/productlines/framework.html (2008)
2. Koznov, D., Romanovsky K.: DocLine: a Method for Software Product Line Documentation Development. Programming and Computer Software, editor V.P. Ivannikov, Vol. 34, №4 (2008)
3. Trujillo, S., Batory, D., Díaz, O.: Feature Refactoring a Multi-Representation Program into a Product Line. In: Proc. of the 5th Int. Conf. on Generative Programming and Component Engineering (2006)
4. Calheiros F., Borba P., Soares S., Nepomuceno V., Vander Alv.: Product Line Variability Refactoring Tool. 1st Workshop on Refactoring Tools, Berlin (2007)
5. Liu J., Batory, D., Lengauer, C.: Feature oriented refactoring of legacy applications. In: Proceedings of the 28th International Conference on Software Engineering, pp. 112–121. ACM Press (2006)
6. Critchlow, M., Dodd, K., Chou, J., and van der Hoek, A.: Refactoring product line architectures. In IWR: Achievements, Challenges, and Effects, pp. 23–26 (2003)
7. Alves V., Gheyi R., Massoni, T., Kulesza, U., Borba P., Lucena, C.: Refactoring Product Lines, In: Proceedings of the 5th international conference on Generative programming and component engineering. Portland, Oregon, USA. pp. 201-210 (2006)
8. Walsh N., Muellner L.: DocBook: The Definitive Guide. O'Reilly (1999)
9. Day, D., Priestley, M., Schell, David A.: Introduction to the Darwin Information Typing Architecture – Toward portable technical information. At: http://www-106.ibm.com/developerworks/xml/library/x-dita1/
10. Parnas, D.: On the Design and Development of Program Families. IEEE Transactions on Software Engineering, March 1976. pp. 1-9 (1976)
11. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley. Boston (2002)
12. Tracz, W.: Collected Overview Reports from the DSSA Project, Technical Report, Loral Federal Systems – Owego (1994)
13. Clements, P.: Being Proactive Pays Off. IEEE Software July/August 2002, pp. 28-31 (2002)
14. Krueger, C.: New Methods in Software Product Line Practice. Communications of The ACM. Vol. 49, №12. pp. 37-40 (2006)
15. Krueger, C.: Eliminating the Adoption Barrier. IEEE Software July/August 2002, pp. 29–31 (2002)
16. Fowler, M., et al.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
17. TeX user group: http://www.tug.org
18. Rockley A., Kostur P., Manning S.: Managing Enterprise Content: A Unified Content Strategy. New Riders (2002)
19. Clark, D.: Rhetoric of Present Single-Sourcing Methodologies. In: SIGDOC'02, Toronto, Ontario, Canada (2002)
20. Albing, B.: Combining Human-Authored and Machine-Generated Software Product Documentation. In: Professional Communication Conference. IEEE Press. pp. 6-11 (2003)
21. Companies using DITA: http://dita.xml.org/deployments
22. Companies using DocBook: http://wiki.docbook.org/topic/WhoUsesDocBook