Evolution Support by Homogeneously Documenting Patterns, Aspects and Traces

Johannes Sametinger
Johannes Kepler University Linz, Austria
sametinger@acm.org

Matthias Riebisch
Technical University of Ilmenau, Germany
matthias.riebisch@tu-ilmenau.de

Abstract

The evolution of complex software systems is promoted by software engineering principles and techniques like separation of concerns, encapsulation, stepwise refinement, and reusability of design solutions. Design patterns capture the expertise for reusable design solutions. Aspect-oriented programming is a methodology that enables the modularization of cross-cutting concerns. Traceability links designate dependencies between requirements, design, and source code. In order to support maintenance, documentation has to enable understandability by describing these issues. Descriptions have to facilitate tool support for automating documentation activities.

In this paper, we use the notion of patterns, aspects and traces for a homogeneous documentation approach. We integrate various types of documentation, keep track of traces from requirements to the source code, keep design information in the source code, and generate additional design views on software systems. We have implemented these ideas as an extension to javadoc, the documentation approach used by Java. This extension can be used to automatically generate views on the design and on aspects as well as on traceability links as part of the standard javadoc system documentation.

Keywords: *evolution, maintenance, documentation, design pattern, traceability, object-oriented design, javadoc*

1. Introduction

The complexity of software systems obstructs their evolution by confusing developers during modifications, causing mistakes and design deficiencies. Managing the complexity of software systems is one of the major challenges to both developers and maintenance personnel. Software engineering principles like hierarchically structuring, information hiding and separation of concerns help to master the complexity. The impact of changes to the architecture, the design, and the implementation has to be understood. Even if only one unit of work, e.g., one concern or aspect, is affected by a particular change, an understanding of interactions with other parts of the system is necessary. Therefore, connections between separated units of work have to be provided.

Design patterns capture the expertise for reusable design solutions [4]. Patterns describe repeatedly occurring problems and their solutions in such a way that these solutions can be reused manifold, without ever doing it the same way twice [1]. Design patterns are essential to maintenance as they provide information about design issues. Explicit information about such design issues can prohibit design blurring and degradation during the maintenance process. Design patterns have gained wide-spread acceptance and use. But despite their definite advantages, there are impediments to pattern-based software engineering. Design patterns are treated only as non-software artifacts. Programmers create, extend, and modify classes throughout the software and tend to lose sight of the original patterns, which may lead to a major maintenance problem.

The principle of separation of concerns helps mastering complexity during design. Aspect-oriented programming is a programming methodology that supports this principle within the implementation by modularization of crosscutting concerns [5]. Aspects provide a bridge between design and implementation. During design, aspects facilitate the thinking about crosscutting concerns as well-defined entities. During implementation, aspects make it possible to program in terms of design aspects [14]. Furthermore, composition techniques for components representing aspects supports flexibility and reusability. Aspects are essential to maintenance for the identification of related facets in the source code.

The evolution of a complex software system requires comprehension about consequences and dependencies. Traceability links designate dependencies between requirements, design, and source code. They improve program comprehension by, e.g., showing the impact of changes due to new features [8]. Traceability represents an important factor enabling evolution. Successful reusability of artefacts, e.g., design, source code components or other, depend on their evolution in order to reach a higher level of maturity and robustness. The product line approach [3]—a methodology for large-scale reuse for families of systems—is supported by documenting traceability links, roles and other dependencies according to the approach explained here. Traceability information ensures that consistency is retained and as much information as possible is available to support the maintenance process.

Documentation is the only tangible way of representing software and its process. It has to be consistent and readable. System documentation describes the implementation including the requirements specification, the system architecture, detailed design descriptions, the source code, test plans, etc. [12]. Hyperlinks facilitates navigation within

complex structures, thus supporting understandability. However, in order to keep documentation up-to-date and consistent, automatic generation and CASE tool integration is needed. Javadoc is a tool that parses declarations and special documentation comments in a software system's source files and produces web pages describing classes, interfaces, methods, and fields for an online, hypertext-based documentation [13]. The content and format of the output can be customized.

In this paper we will provide an integration of the concepts mentioned above by extending the standard documentation. The approach includes important facets of a software system including patterns, aspects and traces, thus including and integrating design views, design documents as well as traces among them. In Section 2, we will give an introduction to basic concepts, including patterns, aspects, traces and javadoc. In Section 3, we present our documentation approach. In Section 4, we describe the implementation using javadoc. Conclusions follow in Section 5.

2. Basic Concepts

Design patterns [4], aspect-oriented programming [5, 14], traceability links [8], and javadoc [13] build the cornerstone of our approach for supporting evolution by extended documentation.

2.1 Patterns

Object-oriented design patterns provide a scheme for describing best practices in the domain of object-oriented design. They are frequently described as a problem/context/solution triple [2,4,7]. "A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context" [2].

Design patterns are abstract ideas that can be illustrated in different ways, for example, by using class diagrams [2], role models [9], or a combination thereof. The choice of a particular modeling technique depends on how well the presentation conveys the pattern idea to its readers [9]. Design patterns provide a common design vocabulary, a documentation and learning aid. Therefore, they are an adjunct to existing methods, and a target for refactoring. Their use typically involves several steps, i.e., reading the documentation for an overview; studying the structure, the participants, and collaborations; understanding the sample code; choosing names for pattern participants that are meaningful in the application context; defining the classes; assigning application-specific names for operations in the pattern; and implementing the operations to carry out the responsibilities and collaborations in the pattern [4].

2.2 Aspects

Multiple concerns have to be considered in the case of complex systems. Even if separated in design, concerns frequently are merged during implementation. Aspect-oriented programming enables the modularization of cross-cutting concerns by providing module composition and interaction concepts, as well as references among them [5, 14]. As a consequence of modularization, changes can be carried out by separate aspect modules. During implementation, the aspects are composed by what is called an aspect weaver [6].

Capturing and documenting aspects is useful even if not using the full power of aspect-oriented programming. We consider source code that is logically belonging together but which is possibly scattered over many locations as aspects. For example, write methods that are spread over many classes may all be used to write a complex data structure to a file.

2.3 Traces

Links between requirements, design, other subsequent models and source code can show the impact of changes to the developer. Most of these connections designate dependencies within the system parts. Such links are often called traceability links. Traceability is useful to support change. Various kinds of traces are important [8]:

- traces between features and their implementation,
- traces of relations between requirements, design decisions, and features.

In the approach of multi-layered rich traceability, links are used to support understandability and change in requirement specification by supplying navigation within a hierarchical structure, among document parts and reports. Enriched multi-layer traceability concepts are used to visualize:

- relations and constraints between user requirements and features including their dependencies,
- constraints within architecture, design and implementation, and
- constraints among features that are relevant for feature configuration.

Additionally, the use of patterns and aspects requires more types of relations between units of work, i.e., roles, composition nodes and instances. They can be represented by traces as well.

Software process definitions, e.g. the CMM [11], demand for documentation at all product levels: requirements, architecture, design, implementation, and installation. In order to keep all these documents consistent and complete during a sequence of iterations and changes, support is necessary at the methodical and at the tool level. The use of

patterns and aspects especially affects architecture, design, and implementation. In order to simplify an update of the documents, an explicit description of constraints between the documents and their elements is needed. This traceability information is added to the models and the relevant documents.

2.4 Javadoc

Javadoc is a tool from Sun Microsystems for generating API documentation out of declarations and documentation comments in Java source code. Javadoc produces HTML documentation describing the packages, classes, interfaces, methods, etc. of a software system.

Javadoc output can be customized by means of doclets. A doclet is a program written with the doclet API that specifies the content and format of the output to be generated. Thus, a doclet can, for example, generate any kind of text file output, such as HTML, SGML, XML, RTF, and MIF. Sun provides a standard doclet for generating HTML format documentation. Doclets can also be used to perform special tasks not related to producing systems documentation. For example, a diagnostic doclet could be created to enable model checking, for example, whether all class members have documentation comments [13]. Javadoc parses special tags embedded within a Java doc comment. These doc tags are used to automatically generate a complete, well formatted API from the source code. All tags start with an "at" sign (@), e.g., @author. The tags are used to add specific information like a method's parameters (@param), return type (@return), and exceptions (@exception), see example in Fig. 1.

```
/**
 * A class representing a simple drawing window on
 * the screen. Example usage:
 * <pre>
 *    SimpleDraw s = new SimpleDraw();
 *    s.setVisible(true);
 * </pre>
 *
 * @author  Johannes Sametinger
 * @version  1.2
 * @see  java.awt.Frame
*/
class SimpleDraw extends Frame {
   ...
}
```

Fig. 1: Javadoc Comment

3. Documentation for Evolution Support

Typically, object-oriented software systems consist of many classes, patterns, aspects, and traces. In practice, documentation contains details of classes. Overview information covering e.g., architectural principles are contained in manually written documents. They are not updated automatically during modifications to the source code. Our aim is to automatically document systems in many respects. In the following sections, we demonstrate how to integrate the documentation of patterns, aspects, and traces into regular system documentation. The extension of javadoc by this kind of links enables an automatic update of documents and their online availability in HTML format.

3.1 Patterns

Design patterns describe relations within an abstract object-oriented model. These relations have to be mapped from a pattern description scheme onto concrete design and code. Roles enable such a mapping. They describe how collaborating objects that play one or more roles achieve a common goal according to a pattern.

In Fig. 2 we can see the start page of the system documentation of an application. In the top left panel we can see the entries “All Aspects”, “All Classes”, “All Patterns”, and “All Traces”. By clicking on the item “All Patterns”, we get a list of patterns in the lower left panel. This panel provides a summary and a list of all patterns in the system. If we click on "Summary" we get the pattern summary as shown in the big right panel in Fig. 2. Here we can see a table with all patterns in the system as well as their type. In the pattern summary of Fig. 2, there are eight design pattern instantiations, one abstract factory, two iterators, four observers, and one visitor.

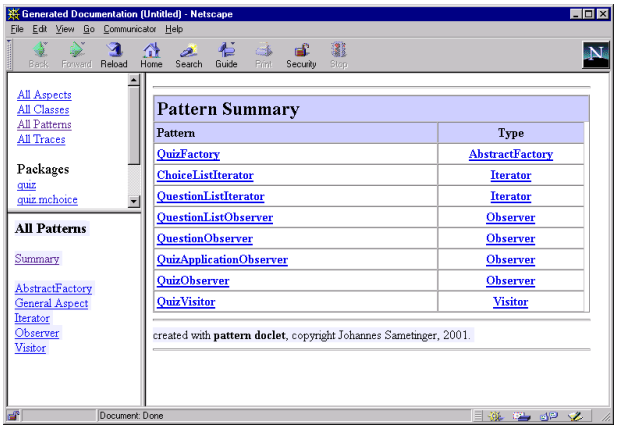

Fig. 2: Pattern Summary

Following a link to any of the patterns in the left column of the summary brings us to more detailed information about a pattern as indicated in Fig. 3. The overview shows the name of a pattern, provides links to general information and lists all roles that have been found. Any role can be played by one or several classes, methods, and/or fields. These are listed for each role together with a short text. The documentation of any of the role players can be directly accessed by following the links that are shown as underlined in Fig. 3.

QuizVisitor
see: Visitor

AbstractVisitor
- interface quiz.Visitor

Client
- writeHTML(BufferedWriter), method in class quiz.QuizApplication
  produces HTML output by means of a visitor

ConcreteVisitor
- class quiz.HTMLVisitor
  We are producing HTML output.
- class quiz.LaTeXVisitor
  We are producing LaTeX output.

DataStructure
- accept(Visitor), method in class quiz.QuestionCatalog
  QuestionCatalog accept visitor
- accept(Visitor), method in class quiz.QuizApplication
  QuizApplication accept visitor
- accept(Visitor), method in class quiz.Question
  this is the abstract method for all Question sub-classes
- accept(Visitor), method in class quiz.vocabulary.VocabularyQuestion
  accept visitor

Fig. 3: Design Pattern Overview

3.2 Aspects

Aspects are meant to clearly capture important design decisions that involve code being scattered throughout the system, i.e., they crosscut the system's functionality [5, 6]. Aspects have been introduced because programming languages do not provide abstraction and composition mechanisms for several design issues, i.e., for all kinds of units a design process breaks a software system into. Aspects provide an important contribution in trying to capture design issues that cannot be adequately expressed otherwise. Aspects cover only specific design aspects, but can be generic in that they can be applied to classes and methods with certain properties.

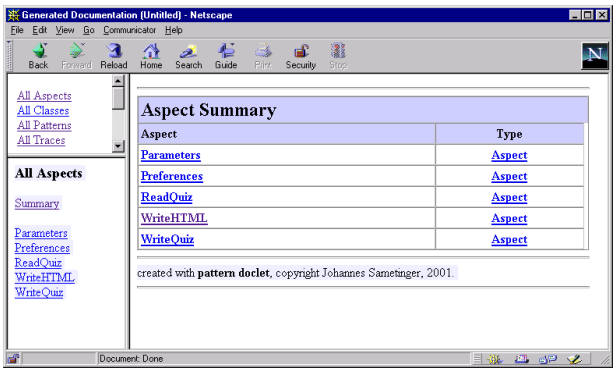

Fig. 4: Aspect Summary

From the understandability point of view, the relations among aspects have to be represented to enable their composition as well as their evolution. For aspects, we introduce the same kind of information about roles and relations as for patterns of a software system, see Fig. 4.

Kiczales et al. distinguish among various forms of aspects, e.g., join points, pointcut designators, advices [6]. These aspect forms can be used to associate code bodies to, say, method calls and, thus, define when the code should be executed. We pursue a much simpler notion of aspects, i.e., we simply document that certain locations in the source code play a role for the performance of a certain task. For example, writing the complex data structure involves method calls of many classes comprising this data structure. The source code for writing this data structure is spread over many locations, and it makes sense, to have an entry point for the access of all these locations. Figure 5 provides this entry point for the aspect of writing a quiz. It lists all the classes and methods involved in the task of writing a quiz. Again, the documentation of any of the players for this aspect can be directly accessed by following the links that are shown as underlined in Fig. 5.

Aspect: WriteQuiz
- save(String, String), method in class quiz.QuizApplication
  here we go saving a quiz.
- write(BufferedWriter), method in class quiz.QuestionCatalog
  here we continue.
- write(BufferedWriter), method in class quiz.QuizApplication
  here we write the quiz.
- write(BufferedWriter), method in class quiz.Question
  here we continue with writing the question.
- write(BufferedWriter), method in class quiz.vocabulary.VocabularyApplication
  here we write the vocabulary quiz.
- write(BufferedWriter), method in class quiz.vocabulary.VocabularyQuestion
  here we continue with writing the vocabulary question.
- write(BufferedWriter), method in class quiz.mchoice.MchoiceChoice
  here we continue with mchoice choice.

Fig. 5: WriteQuiz Aspect

3.3 Traces

Modifications and extensions as well as the reuse of existing software systems require comprehension. The impact of changes to both design and implementation have to be well understood. Such impacts can be made visible by traceability links among documents of software systems, i.e., requirements, design, models, and the source code. Traceability is to be shown as relations among requirements, design decisions, features, and their implementation

In Fig. 6 we can see a list of traces on the left side and a summary on the right side. Traces can be organized hierarchically. For each of them we need a link to the documentation, e.g., to the description of a requirement, and a link to the appropriate source code where the requirement is being implemented. Missing links to the source code indicate either that the requirement had not been implemented yet or that the documentation is not complete yet, i.e., the link between requirement and source code had not been included yet.

Links to the source code can involve classes or methods. This will mostly be the case for rather detailed requirements. Often, fulfilling a certain requirement cannot be done by a single portion of the source code. We can either provide links to all relevant locations in the source code or, what we prefer, to a separate aspect covering this source code. As a consequence, a single link points to an aspect as a cluster of source code rather than source code that is possibly spread over many locations of a solution.

The acquisition of traceability information is not covered by existing development process models. However, dependencies can be derived following the flow of the development processes. They can be elicited by recording design decisions during forward engineering. In our experience, most design decisions during the usual engineering activities are implicitly made by developers, without paying special attention to them. Recording them is a demanding task. Recording fact-based decisions documented in products of high-maturity development processes, e.g., according to higher CMM levels [11], is much easier. Furthermore, reverse engineering activities directly result in the discovery of design decisions.

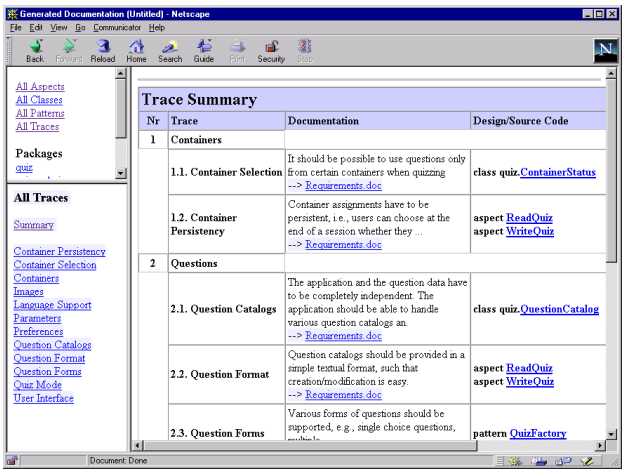

Fig. 6: Trace Summary

3.4 Source Code Integration

There have to be links from the regular system documentation, i.e., documentation generated by standard javadoc, to this extra documentation about patterns, aspects and traces, wherever a class, method or field is involved. In Fig. 7 we can see the documentation for the class HTMLVisitor where the entry "Patterns:" gives all the information about any roles played by this class in a particular design pattern. Again, links can be followed to get information about the pattern instantiation of which the class plays a role, as well as about the design pattern itself.

4. Implementation

A basic infrastructure is indispensable in order to present a system's patterns, aspects and traces in appropriate form to development and maintenance personnel. Tool support has to be provided at the source code level. That means that links to and information about patterns, aspects and traces are kept in the source code, e.g., by means of special comments. All other development activities and tools operating on the source code can make use of the extra information. For example, design tools are able to extract design views out of the source, to present it, and to allow design modifications, supporting appropriate changes in the source code. All changes affecting the source code are resulting in changes to the extra information as well. Tags as targets for links are changed appropriately. Documenting patterns, aspects and traces in the source code can easily be done by means of comments like that used for javadoc, where comments contain information about name, type, and role of an aspect or pattern. This information can be used to recreate design information, as had been outlined in the previous section.

quiz
Class HTMLVisitor
`java.lang.Object`
`  |`
`  +--quiz.HTMLVisitor`

public class **HTMLVisitor**
implements Visitor

HTMLVisitor contains all methods needed for HTML output creation.

Patterns:
- **Visitor**,
  instance: **QuizVisitor**,
  role: **ConcreteVisitor**
  We are producing HTML output.

. . .

Fig. 7: Class Overview

We keep most of the information in the source code and have defined additional tags to standard javadoc. Our tags will be described in Section 4.1. In Section 4.2 we provide an example for the usage of these tags.

4.1 Javadoc Tags

In order to describe design patterns, aspects and traceability links in source code, we introduce new tags that are used similar to any other tag for javadoc. They can be used for classes, methods and fields. For the description of design patterns we use the @pattern tag. The syntax is as follows:

```
@pattern  <pattern name>.<instance name>.
          <role name>  <text>
```

<pattern name> specifies the name of the pattern, e.g., Aspect, Iterator, Visitor. <instance name> specifies the name of the specific instance, e.g., QuestionIterator, FrameIterator. With the name of an instance we can distinguish various instances of the same pattern, e.g., there are two iterator instances named ChoiceListIterator and QuestionListIterator in Fig. 2. <role name> specifies the name of the role played by a class, method, or field, e.g., AbstractVisitor, ConcreteVisitor. <text> is optional and provides additional information to appear in the documentation.

In order to document aspects and traces, we use the tags @aspect and @trace. These tags can also be used for classes, methods and fields. The syntax is as follows:

```
@aspect <name>  <text>
@trace <name>  <text>
```

We have implemented a doclet that processes the @pattern, @aspect and @trace tags and produces documentation output similar to the standard javadoc doclet with additional information as demonstrated in the previous sections.

Javadoc tags are being used in the source code only. In order to support traces, we have added links from and to other documents also. There are two possibilities to provide such connections. First, we can include tags within other documents, e.g., by using hidden text or fields in MS Word documents, or by explicitly stating them in simple text documents. This form of inclusion requires that the document be parsed like the source code and the gathered information about found tags be included in the generated documentation. In the case of design documents, identifiers are used as link targets, and the UML's built-in extensions tagged values are used for describing links [10]. In the case that we if are not be able to include tags to other documents easily, then we use a connector document. Such a document can specify tags and the documents where they belong to.

4.2 Example

We will demonstrate how to use @pattern tags for the documentation of a Visitor pattern. According to this pattern, an abstract visitor, a concrete visitor, an abstract element, concrete elements, and an object structure are the participants [4]. We can use these names for the roles played in the pattern, which we recommend, but we may also use arbitrary names. Figures 8 to 10 show how the participants of the visitor pattern are being identified by means of the @pattern tag.

```
package quiz;
/** represents an operation to be performed on
 * elements of a quiz
 * @author Johannes Sametinger
 * @pattern Visitor.QuizVisitor.AbstractVisitor
 *  declares a visit operation for each class in quiz
 */
public interface Visitor {
        ...
}
```

Fig. 8: Abstract Visitor

All these tags start with the name Visitor, indicating that the Visitor pattern is being documented. The second name QuizVisitor indicates the instance name of this pattern, thus, indicating that all three classes not only play a role in a Visitor pattern, but also play this role in the same instance. There may be several instance of the same pattern in a software system. The interface Visitor in Fig. 8 plays the role of an abstract visitor, class HTMLVisitor in Fig. 9 is a concrete visitor producing HTML output, and class Question in Fig. 10 is an abstract element providing an accept operation and taking a visitor as argument.

```
package quiz;
/** produces HTML output of a quiz
 * @author Johannes Sametinger
 * @pattern Visitor.QuizVisitor.ConcreteVisitor.
 *      We are producing HTML output.
 */
public class HTMLVisitor implements Visitor {
        ...
}
```

Fig. 9: Concrete Visitor

@aspect and @trace tags are used in a similar way. For traces we additionally need links from and to documents other than source code. We are currently experimenting on how to best realize such links.

```
package quiz;
/** represents the basic structure of quiz questions
 * @author Johannes Sametinger
 * @pattern Observer.QuestionObserver.Subject.
 * @pattern Iterator.QuestionListIterator.Item
 */
abstract public class Question extends Observable {
        ...
/** defines an accept operation that takes a visitor
 * as an argument.
 * @param v Visitor to be accepted
 * @pattern Visitor.QuizVisitor.AbstractElement.
 *    this is the abstract method for all Question
 *    subclasses
 */
   abstract public void accept(Visitor v);
}
```

Fig. 10: Visitor's Data Structure

5. Conclusion

We have presented a supporting methodology for evolving software systems by a homogeneously extended documentation approach for patterns, aspects and traces. We keep information in the source code and provide links to other documents. We use an extension to javadoc to generate HTML documentation. The generated documentation includes regular class and method information plus summaries and descriptions of patterns, aspects and traces with links to and from the regular system documentation.

Utilization of this additional information supports comprehension, such that modifications are made consistently and according to requirements, architecture and design. We have implemented javadoc support for source code, including patterns, aspects and traces. Currently, we are working on the support of documents other than source code also, e.g., specification documents in a word processing format. For a next step, we plan to integrate an aspect-based product line composition toolkit.

6. References

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel, *A Pattern Language*, Oxford University Press, New York, 1977.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture*, Wiley & Sons, 1996.

[3] L.M. Northrop, *A Framework for Software Product Line Practice*, Version 3.0, October 24, 2001. available at http://www.sei.cmu.edu/plp/framework.html

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns. Elements of Reusable Object-oriented Software*, Addison-Wesley, 1995.

[5] T. Elrad, R.E. Filman, A. Bader (eds.), "Aspect-Oriented Programming", *Communications of the ACM*, Vol. 44 , No. 10. October 2001.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, "An Overview of AspectJ", *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Hungary*. June 2001.

[7] W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

[8] Matthias Riebisch, Ilka Philippow, "Evolution of Product Lines Using Traceability", *OOPSLA 2001 Workshop on Engineering Complex Object-Oriented Systems for Evolution*, Tampa Bay, Florida, USA, October 15th 2001.

[9] L. Rising (ed.), *The Patterns Handbook: Techniques, Strategies, and Applications*, Cambridge University Press. 1998.

[10] J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[11] Software Engineering Institute (eds.), *Software Process Maturity Questionnare, Capability Maturity Model*, Version 1.1.0. Software Engineering Institute, Pittsburgh, April 1994.

[12] I. Sommerville, *Software Engineering*, 6th edition, Addison Wesley, 2000.

[13] Sun Microsystems, *Javadoc Tool Home Page*, http://java.sun.com/j2se/javadoc/

[14] WWW, *Aspect-Oriented Software Development*, http://aosd.net/