Towards a Standardized Specification Framework for Component Development, Discovery, and Configuration

Sven Overhage

Augsburg University,
Dept. of Application Engineering and
Business Information Systems,
Universitätsstr. 16, 86135 Augsburg, Germany
sven.overhage@wiwi.uni-augsburg.de

Abstract. This paper proposes a standardized framework for the specification of components, which focuses on providing the information necessary to facilitate component development, discovery, as well as configuration. To be applicable in all these domains it predetermines a mix of different specification aspects and thereby prefers well-established formal notations to denote specifications. The framework orders component specifications using a thematic grouping into different pages, which are respectively subdivided into aspects. Currently, it provides general and commercial information (white pages), component classifications (yellow pages), as well as information about the component functionality (blue pages), the external component structure (green pages), and the quality of the component implementation (grey pages). The blue pages focus on providing a lexicon of implemented domain-related concepts (the domain model), the green pages comprise contractually specified interfaces, and the grey pages contain extra-functional attributes according to the ISO 9126 quality framework.

1 Introduction

Component-based software engineering promises to contribute a lot of advantages to the field of today's software engineering [10], [22] and is based on a rather simple-sounding central paradigm: new applications are being developed by browsing component catalogues (repositories), discovering appropriate existing components, and configuring those components using connectors (and, perhaps, some glue code) [10], [22]. Despite all the frequently mentioned advantages, however, component-based software engineering encountered many obstacles that prevented its break-through until now. Among those, especially identifying and managing heterogeneities between components during the process of application development turned out to be main issues [12].

Heterogeneities reduce the compatibility between components so that at least an adapter has to be designed in order to enable interactions between them. In theory, they can be avoided by establishing (generally accepted) standards which have to be applied during component development, e.g. architectural and content-related standards. However, it is likely that even such standards hardly manage to completely prohibit the emergence of heterogeneities in practice, since providing superior component designs usually is a competitive advantage for component producers [22]. Identifying heterogeneities between components (during component discovery) and managing them accordingly (during component configuration) are highly demanding tasks and thus have the potential to thoroughly complicate the simple sounding paradigm of component-based application development. Often, both have to be based on exhaustive (costly) evaluation, component testing or even reverse-engineering of components [12], [15], [26], which is inappropriate and, moreover, almost impossible for black-box components that prohibit any form of source code review [26].

Thus, providing a substantiated methodology to support component discovery and configuration becomes a critical success-factor of component-based software engineering. It should make use of a specification framework that provides explicit information about the external view of a component and its context dependencies. Once equipped with adequate component specifications, composers are put

into the position to efficiently decide whether a set of components can be configured, to predict important characteristics of the resulting configuration, and to estimate the required efforts for configuration (e.g. due to adapter development). Moreover, a specification framework is an important prerequisite for the development of tools and processes that support (automate) component discovery and configuration.

Establishing a *standardized* framework moreover enforces the development of interoperable (compatible) tools and processes. Standardizing component specifications is a popular practice in other (more advanced) engineering disciplines that already have developed component catalogues and tools for configuration (construction) [1]. This position paper proposes a specification framework for software components based on the experiences made during a standardization process devoted to the specification of business components within the German Society of Informatics [1]. It should be looked upon as a suggestion to start standardization within the component community.

2 Related Work and Applications

Currently there are only very few specification frameworks that have been standardized and introduced into the practice of component-based software engineering. One of those is a framework specialized to describe the external view of XML Web services, a special sort of "hosted" components [23] that can be invoked using an XML based remote procedure call. This specification framework is a part of the UDDI (Universal Description, Discovery, and Integration [24]) standard which belongs to the emerging Web service architecture [3].

Nevertheless, component specifications are widely acknowledged to be of key importance and have already been addressed by various authors, who are proposing specialized specifications, e.g. to specify technical properties and interface contracts [1], [5], [6], [10], [14], [22], to classify components using enumerated and faceted classification schemes [20], to provide information about commercial aspects and component acquisition [1], [25], or to store components within component libraries (repositories and marketplaces) [1], [2], [25]. The specification of technical and commercial aspects of components especially supports component assessment (during discovery) and automated adapter/connector (stub) generation (during configuration), while storing and classifying components within component libraries mainly facilitates the search for applicable components (during discovery). Moreover, component specifications are often used to denote requirements and to guide component development [6], [10]. Finally, specification frameworks are an important methodical basis for tools and processes which support component discovery by means of multi-criteria decision-making, artificial intelligence, and specification matching (an overview is given in [21]) – they usually build upon the structure of a specification framework (e.g. as goal-hierarchy) and use component specifications as input to assess the applicability of components in an application development context.

3 Principal Requirements

A component specification framework predetermines a set of specifications and, in general, should fulfil the following requirements:

- Specifications ought to be **methodically substantiated**, which means that each specification should be justified by relating its emergence to corresponding development phases and providing utilizations for the purposes of software development.
- The framework should **provide a complete documentation** of a component. When applying the specification framework to a certain field of application, one should be able to perform the corresponding tasks solely on the basis of the provided specifications.
- The framework has to be **normative** and precisely determine what is to be specified (the specification extent) and which notations have to be used to denote specifications (the specification formats) to enforce homogeneous component specifications that can be utilized by CASE tools.

– The framework structure ought to be **modular and extensible**. Modularizing component specifications, which usually have different utilizations and refer to different target groups, reduces complexity and enhances readability. Moreover, this supports future (backward-compatible) extensions which can be achieved simply by adding new modules.
– The framework should **support different component types**. Although today's standardized specification frameworks (e.g. UDDI [24]) usually support components based on a special technology only, their overall structure is platform-independent and thus justifies the development of a unique specification framework.
– The framework ought to provide both **human- and machine-understandable specifications**. In order to enable tool-supported or automated discovery and configuration, formal (machine-understandable) specifications should be preferred compared to informal (natural) languages. On the other hand, specifications should be simple enough for a human programmer to use.

4 The Unified Software Component Specifications Framework

Based on the pre-mentioned principal requirements, this chapter focuses on elaborating a specification framework to support component development, discovery, and configuration. It is called "Unified Software Component Specifications" (USCS) and has been designed to support multiple component technologies. Additionally, it maintains backward-compatibility to the standardized UDDI specification framework [24], which can be augmented to contain all of the here-mentioned specification aspects [19].

The framework theoretically justifies relevant specifications by using a classification schema (see figure 1), which distinguishes three significant perspectives to completely specify the external view of an application (part) [7], [9], [10], [13], [17]: the static view concentrates on the data model (i.e. the structure of data and relationships between data elements) and quality attributes, which refer to the component as a whole and remain constant during the lifecycle. The operations view focuses on the operations model (i.e. the structure of operations and relationships between operations) as well as operation-specific quality attributes. Finally, the dynamic view covers the process model (i.e. relevant control flows, orders, and causal dependencies between operations), which is augmented with the corresponding performance characteristics.[1]

	Functionality / Concepts (domain-related)	External Structure / Interfaces (logical)	Implementation / Quality (physical)
Static View	Objects (Data Model)	Data-Types	Usability, Maintainability
Operations View	Tasks (Operations Model)	Events, Methods, Assertions	Functionality, Reliability
Dynamic View	Processes (Flow Model)	Coordination Constraints	Efficiency

Fig. 1. The initial structure of the component specification framework is determined by a classification schema.

Moreover (and orthogonal to that), the framework distinguishes component specifications according to their content and their emergence during the process of component (or application) development [10]: specifications describing the component functionality usually originate from domain engineering and are provided as a set of domain-related concepts (either in the form of a lexicon or a domain model, respectively) [6], [8], [10]. The (external) component structure typically is determined during architectural design and consists of one or more component interfaces [6], [10]. Finally, extra-functional in-

[1] Operations and dynamic view are often summarized and commonly addressed as behavior.

formation about the component implementation, which is available after the corresponding development phase has been completed, is provided in the form of ISO 9126 quality attributes. Figure 1 shows the resulting classification schema and uses it to identify different classes of specifications that will be integrated into the specification framework.

The identified specification classes can be used to describe the provided and required services of a component. In so doing, the specification of the external component structure contains the provided and required interfaces [6], [10]. Accordingly, both the description of the component functionality and the implementation quality can respectively be partitioned into provided as well as required functionality/quality.

Taking into consideration these theoretical foundations, the proposed specification framework introduces an aspect-oriented structure in order to make use of different specialized (and preferably well-established) notations and to support future extensibility. Currently, it introduces a total of eleven aspects in order to attempt a complete component specification (see figure 2). They are structured using a thematic grouping into different pages. White pages provide general and commercial information about components. Yellow pages contain both enumerated and faceted component classifications. Blue pages hold information about the component functionality in the form of a component domain lexicon (which comprises domain-related concepts). Green pages provide contractually specified interfaces and, finally, grey pages contribute extra-functional information about the quality of the component implementation.

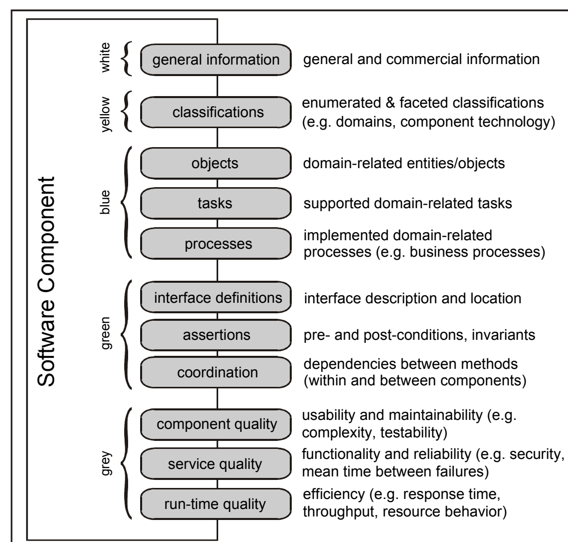

Fig. 2. The Unified Software Component Specifications Framework consists of eleven aspects.

Every aspect comprises a principal format to denote specifications and thereby prefers formal notations compared to natural language to enforce precise and machine-understandable specifications. Each notation is platform-independently applicable to specify components. In addition, a specialized mix of notations, which have already been established to specify XML Web services, has been tied together to maintain compatibility with the emerging Web services architecture [3]. This specialized specification framework is called WS-Specification and discussed in detail in [19].

4.1 White Pages: General and Commercial Information

The white pages contain general and commercial information about a component which is required to store and exchange components. General information consists of the component name, its unique identifier, version, description, producer, administrative contacts, and dependencies to other components. Commercial information refers both to the conditions of purchase and usage.

Specifications referring to the conditions of purchase describe supported distribution channels that respectively consist of a distribution form, a price, accepted payments, and the scope of supply (i.e. a list of artifacts included with the component shipping). The conditions of usage are determined by a license agreement.

The preferred notation for these rather descriptive specifications is natural language. To support precise and machine-understandable specifications, the specification framework provides taxonomies, which have either directly to be used for specification or to classify specifications containing arbitrary text. Figure 3 illustrates the use of taxonomies and shows an example specification.

Name Oversoft EasyBookKeeping
Version 1.0.3455
Identifier 206B2F65-E7BC-47a0-8DCF-4E34347322AA
Dependencies Oversoft ProfitAndLossAccount 1.0 (EA517DD8-CAC5-44e7-B560-4280FAF2FB77)
Producer **Name**: Oversoft Software **Means of Communication** (type: e-mail): info@oversoft.biz **Means of Communication** (type: phone): +49 6003 810 260 **Address line** (type: city): Rosbach **Address line** (type: country): Germany
Distribution Channel **Identifier**: Internet download **Price** (type: USD): 399.90 **Accepted Payment** (type: credit card) **Scope of Supply**: easybookkeeping.cab, handbook.doc, setup.exe
Terms of use **Specification** (type: license agreement): Oversoft EULA …

Fig. 3. An example specification illustrates the white pages and the use of taxonomies during specification.

4.2 Yellow Pages: Classifications

Enumerated and faceted component classifications [20] are grouped within the so-called yellow pages. They are essential to specify the (horizontal or vertical) domain which a component belongs to. The specification framework therefore provides a variety of standardized taxonomies, e.g. UNSPSC or NAICS to describe the domain from an economical perspective. Moreover, the yellow pages contain information about the underlying architecture and technology of a component. To support the specification of these attributes, the framework provides taxonomies, which list implementation technologies (e.g. EJB, COM, .NET, XML Web service, CCM etc.), conceptual component types (e.g. specialized component, framework, application etc.), and reuse concepts (e.g. logical reuse, which means "provided as redistributable", and physical reuse, which means "provided as remote service"). Figure 4 shows an example.

Domain (type: UNSPSC – Accounting and Controlling)
Component type (type: specialized component) **Reuse concept** (type: logical) **Technology** (type: Microsoft .NET Framework 1.0)

Fig. 4. An example specification illustrates the yellow pages.

Additionally, yellow pages can be used to classify the location of mobile services (which form a special component type), e.g. by providing a standardized geographic taxonomy like the Microsoft GEO

taxonomy that originally belongs to the UDDI framework [24].

4.3 Blue Pages: Domain-Related Information about the Functionality

The blue pages summarize domain-related information about the component functionality that was gathered during the domain design [6], [8], [10] and later on used during the implementation of a component. The information is provided as a domain lexicon (see figure 5) that contains relevant concepts, definitions, and relationships between concepts. It distinguishes three kinds of concepts: objects (entities), operations (tasks), and processes. Concepts can principally be related to each other by using one of the fundamental relationship types: abstractions or compositions. Abstractions determine a certain degree of identity between concepts and can be used to execute compatibility tests and perform subtyping between components. Typical abstractions are the is-synonym-to (is-identical-to), is-specialization-of, and is-generalization-of relationships. Compositions are used to combine concepts and to express the (compound) structure of concepts. Typical compositions are order relationships, the is-part-of, and consist-of relationship. The framework provides a specialized taxonomy to express a variety of different relationships between concepts.

Concept (type: entity): BALANCE

- Short definition: BALANCE $=_{DF}$ the comparison of ASSETS and LIABILITIES of a company at a special date (CUTOFF DATE) based on a LEGAL REGULATION.
- Relationships: US-GAAP_BALANCE is a BALANCE.
  IAS_BALANCE is a BALANCE.

Concept (type: task): ANNUAL ACCOUNTING

- Short definition: ANNUAL ACCOUNTING $=_{DF}$ …
- Relationships: ANNUAL ACCOUNTING consists of CLOSE ACCOUNTS.
  ANNUAL ACCOUNTING consists of BALANCING.
  ANNUAL ACCOUNTING consists of PROFIT & LOSS ACCOUNTING.

Concept (type: process): EXECUTE ANNUAL ACCOUNT

- Short definition: EXECUTE ANNUAL ACCOUNT $=_{DF}$ …
- Relationships: EXECUTE ANNUAL ACCOUNT starts with CLOSE ACCOUNTS.
  CLOSE ACCOUNTS is sequentially followed by PROFIT & LOSS ACCOUNTING.
  PROFIT & LOSS ACCOUNTING is sequentially followed by BALANCING.

Fig. 5. An example specification illustrates the blue pages.

Concept definitions give an impression on what a component or an interface-method does. Explicitly specifying implemented concepts and relationships between them facilitates the emergence of domain-specific standards and provides valuable input for component development, assessment, and configuration. Since they usually can be mapped to data types, properties, methods, events, or coordination constraints, concept definitions moreover assist in both the identification of heterogeneities and the design of adapters during configuration.

The principal notation for domain-related information is normative language, a special format to denote an ontology [18] that is both machine- and human-understandable. It uses a standardized form of natural language for specification and provides patterns to build sentences (e.g. A is a B etc.). Alternatively other ontology-based notations could be used. However, most of them are rather complex and render both specification and readability a lot more difficult.

4.4 Green Pages: Contractually Specified Interfaces

The green pages provide information referring to a component's interfaces which is necessary to correctly configure and finally invoke component services. They contain both the provided and the required interfaces. In addition, interfaces that support component customization (e.g. by providing parameters) can be specified. Each interface consists of different definitions, including named interface-

methods and their signature, named public properties as well as variables, constants, declarations of specific data types, possible exceptions and events (see figure 6). The principal notation to specify interface-definitions is OMG IDL, which supports specifying provided, required, and customization interfaces.

```
interface EasyBookKeeping {

    typedef string accountno;
    typedef double quantity;

    struct account {
        accountno   n;
        quantity    safetyquantity;
        quantity    reorderimgquantity;
    };
    struct booking {
        accountno   n;
        date        executiondate;
        string      orderno;
        double      bookingquantity;
    };

    exception toolittlequantity {};

    void            book(in booking b);
    void            reserve(in booking b) raises (toolittlequantity);
    quantity        calculatequantityfor(in accountno n, in date z);
    quantity        calculatedemand(in accountno n, in date z);
};

interface ProfitAndLossAccounting {

    typedef double quantity;

    struct account {
        accountno   n;
        quantity    safetyquantity;
        quantity    reorderimgquantity;
    };

    quantity executeProfitAndLossAccount (in account[] input);
};
```

Fig. 6. An example illustrates the specification of provided and required interfaces.

In addition, the specification of invariants and pre- and post-conditions for each of the interface-methods is supported to enable designing applications by contract [16]. Pre-conditions express the constraints under which an invoked method returns correct results. Accordingly, post-conditions describe the respective state resulting from a method's execution und thus guarantee that it will satisfy certain conditions (provided that it was called with the pre-condition satisfied). The principal notation to specify these so-called assertions is the Object Constraint Language (OCL), a formal notation provided as part of the Unified Modeling Language (UML). An OCL expression firstly defines the context of a specification by linking it to an interface-method or a component. Thereafter, the respective assertions that apply to the context are listed.

```
EasyBookKeeping
    self.account->forall(k:account | k.safetyquantity >= 0)

EasyBookKeeping::calculatequantityfor(n:accountno,z:date):quantity
    pre:   self.account->exists(k:account | k.accountno = n)
    post:  result = self.booking->iterate(b:booking; r:quantity = 0 |
           if  b.accountno = n and b.date <= z
           then
               r + b.bookingquantity
           endif
           )
```

Fig. 7. An example specification illustrates the specification of assertions.

Figure 7 contains an example specification. The first expression states that the security quantity for each account has to be greater or equal zero. The second expression determines that the service *calculatequantityfor* can only be executed for an account known to the component. Moreover, the quantity of an account (returned by the service *calculatequantityfor)* is the sum of the quantities that have been booked on this account up to the current date (decreases are considered as negative quantities and thus

subtracted).

Finally, the framework supports specifying constraints referring to the ordered invocation of interface-methods (coordination constraints), which can both occur within an interface and between interfaces. Usually, interface-methods may not be arbitrarily invoked and have to be called in a predetermined order that is difficult to identify solely on the basis of interface-definitions. Thus, coordination constraints provide valuable information which can be utilized during configuration to determine the resulting control flow of the application. They can be denoted using a specialized format that is based on OCL and extends it with temporal operators to implement a temporal logic (for a complete list of temporal operators see [1]). Extending a well-established specification format instead of creating a new one enhances both acceptance and the probability for future tool-support.

Figure 8 shows an example specification expressing that the *book*ing service may only be called after the *reserve* method, which debits the account so that the booking can take place.

```
EasyBookKeeping::book(b:booking)
    pre:   sometime_past(reserve(b:booking))
```

Fig. 8. An example specification illustrates the specification of coordination constraints.

4.5 Grey Pages: Information Referring to the Implementation

The grey pages hold information about the component implementation. Although this may sound somewhat strange at a first glimpse (components are black-boxes), providing extra-functional information about the implementation quality nevertheless is useful since it serves as valuable input for component discovery. Quality attributes can refer both to the component as a whole as well as to single interface-methods. Following an adoption of the ISO 9126 quality model for COTS components [4], the specification framework supports specifications referring to the usability, maintainability, functionality, reliability, and efficiency of a component implementation.

```
type Efficiency = contract {
    responseTime: decreasing numeric msec;
    componentSize: decreasing numeric kbyte;
}
```

```
qualityProfile for EasyBookKeeping = profile {
    from book require Efficiency contract {
        responseTime < 5 msec;
        componentSize == 1428 kbyte;
    }
}
```

Fig. 9. An illustration of the USCS type library and the specification of quality attributes within the grey pages.

The principal notation to denote quality specifications is QML [11], which supports both the definition of specific quality attributes along with their respective metrics as QML contract types and the specification of concrete implementation quality characteristics as QML contracts. To ensure the specification of homogeneous quality characteristics, the specification framework provides a QML type library with quality attributes and metrics (based on ISO 9126 and [4]), which can be augmented with additional quality attributes as needed (see figure 9).

5 Future Directions

The proposed specification framework has recently been evaluated in practise (initial case studies can be found under www.fachkomponenten.de). Ongoing research focuses on introducing a standardized data model that can be used to store and exchange specifications between tools. This data model will also be used to integrate different notations (e.g. WS-Specification and USCS) via a model-view con-

cept. Furthermore, UML 2.0 is currently being introduced into the specification framework as an alternative mix of graphical notations.

Based on the proposed specification framework various CASE tools and processes are currently being developed, among those a repository to store and exchange components, a marketplace to trade components, a tool to support the specification-process, a flexible process model for component-based software engineering, and a process for component discovery based on multi-criteria decision-making.

An important goal is to start standardization of component specifications at an international level. This framework has been developed to contribute an initial proposal and open discussions in a broader standardization interest group.

References

1. Ackermann, J., Brinkop, F., Conrad, S., Fettke, P., Frick, A., Glistau, E., Jaekel, H., Kotlar, O., Loos, P., Mrech, H., Ortner, E., Overhage, S., Raape, U., Sahm, S., Schmietendorf, A., Teschke, T., Turowski, K.: Standardized Specification of Business Components. German Society of Informatics (2002) http://wi2.wiwi.uni-augsburg.de/fachkomponenten/memorandum.php?lang=eng
2. Apperly, H.: Configuration Management and Component Libraries. In: Councill, W. T., Heineman, G. T. (eds.): Component-Based Software Engineering: Putting the Pieces Together. Addison Wesley, Upper Saddle River, New Jersey (2001): 513–526
3. Austin, D., Barbir, A., Garg, S. (eds.): Web Service Architecture Requirements. W3C Working Draft (2002) http://www.w3.org/TR/wsa-reqs
4. Bertoa, M. F., Vallecillo, A.: Quality Attributes for COTS Components. In: Proceedings of the 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002) http://alarcos.inf-cr.uclm.es/qaoose2002/docs/QAOOSE-Ber-Val.pdf
5. Beugnard, A., Jézéquel, J.-M., Plouzeau, N., Watkins, D.: Making Components Contract Aware. In: IEEE Computer 32 (1999) 7: 38–45
6. Cheesman, J., Daniels, J.: UML Components: A Simple Process for Specifying Component-Based Software. Addison Wesley, Upper Saddle River, New Jersey (2000)
7. Cook, S., Daniels, J.: Designing Object Systems. Object-Oriented Modeling with Syntropy. Prentice Hall, Englewood Cliffs (1994)
8. Czarnecki, K., Eisenecker, U. W.: Generative Programming: Methods, Tools, and Applications. Addison Wesley, Upper Saddle River, New Jersey (2000)
9. Davis, A. M.: Software Requirements: Objects, Functions, and States. Prentice Hall, Englewood Cliffs (1993)
10. D'Souza, D., Wills, A. C.: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison Wesley, Reading, Massachusetts (1999)
11. Frolund, S., Koistinen, J.: QML: A Language for Quality of Service Specification. Technical Report HPL-98-10, Hewlett-Packard Laboratories (1998)
12. Garlan, D., Allen, R., Ockerbloom, J.: Architectural Mismatch: Why Reuse Is So Hard. In: IEEE Software 12 (1995) 6: 17–26
13. Graham, I.: Migrating to Object Technology. Addison Wesley, Wokingham (1994)
14. Kiniry, J. R.: Leading to a Kind Description Language: Thoughts on Component Specification. Caltech Technical Report CS-TR-99-04, California Institute of Technology (1999)
15. Kontio, J.: A Case Study in Applying a Systematic Method for COTS Selection. In: Proceedings, 18th International Conference on Software Engineering (ICSE). IEEE Computer Society Press (1996): 201–209
16. Meyer, B.: Object-Oriented Software Construction. Prentice Hall, Cambridge (1988)
17. Olle, T. W., Hagelstein, J., MacDonald, I. G., Rolland, C.: Information Systems Methodologies: A Framework for Understanding. Addison Wesley, Wokingham (1991)
18. Ortner, E., Schienmann, B.: Normative Language Approach – A Framework for Understanding. In: Thalheim, B. (ed.): Conceptual Modeling – ER '96, 15th International Conference on Conceptual Modeling, Proceedings, Cottbus 1996. Springer, Berlin (1996): 261–276
19. Overhage, S., Thomas, P.: WS-Specification: Specifying Web Services Using UDDI Improvements. In: Chaudri, A. B., Jeckle, M., Rahm, E., Unland, R. (eds.): Web, Web Services, and Database Systems. Lecture Notes in Computer Science (LNCS 2593), Springer, Berlin (2003): 100–118
20. Prieto-Diáz, R.: Implementing Faceted Classification for Software Reuse. In: Communications of the ACM 34 (1991) 5: 89–97
21. Ruhe, G.: Intelligent Support for Selection of COTS Products. In: Chaudri, A. B., Jeckle, M., Rahm, E., Unland, R. (eds.): Web, Web Services, and Database Systems. Lecture Notes in Computer Science (LNCS 2593), Springer, Berlin (2003): 34–45
22. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison Wesley, Harlow (1998)
23. Szyperski, C.: Components and Web Services. In: Software Development Magazine 9 (2001) 8
24. UDDI Organization (ed.): UDDI Executive White Paper. UDDI Standards Organization Public Draft (2001) http://www.uddi.org/pubs/UDDI_Executive_White_Paper.pdf
25. Varadarajan, S., Kumar, A., Deepak, G., Pankaj, J.: ComponentXChange: An E-Exchange for Software Components. http://www.cse.iitk.ac.in/users/deepak/papers/ComponentXchange.pdf
26. Weyuker, E. J.: The Trouble with Testing Components. In: Councill, W. T., Heineman, G. T. (eds.): Component-Based Software Engineering: Putting the Pieces Together. Addison Wesley, Upper Saddle River, New Jersey (2001): 499–512