Facilitating Reuse of Code Checking Rules in Static Code Analysis

Vladimir A. Shekhovtsov, Yuriy Tomilko, and Mikhail D. Godlevskiy

Department of Computer-Aided Management Systems,
National Technical University "Kharkiv Polytechnical Institute", Ukraine
shekvl@yahoo.com, tomilko.yuriy@mail.ru, god_asu@kpi.kharkov.ua

Abstract. Currently, the rationale of applying code checking rules in static code analysis is often not captured explicitly which leads to the problems of rule reuse in similar development contexts. In this paper, we investigate the process of tracing possible sources of such rules back to design decisions and quality requirements. We present an idea of storing the rationale information along with particular code checking rules in a rule repository. We argue that such information is related to particular design decisions or patterns that need to be enforced by the rule and to generic properties of these decisions such as corresponding quality characteristics. We show how a reuse support tool with underlying rule repository can aid in defining the recommended set of rules to be reused while making recurring design decisions or applying design patterns.

1 Introduction

Static code analysis is a special kind of inspection [17] which is performed over the source (or bytecode-compiled) code with a goal of revealing and correcting errors introduced during software development [1, 33]. It also helps assessing software quality attributes and checking standards compliance. Specific static analysis tools [29, 31] (such as FxCop, Gendarme, NDepend, FindBugs, PMD etc.) allow the user to apply the set of *code checking rules* to the code looking for known bugs, various violations of common recommendations and policies, other code deficiencies. Applied properly, static code analysis tools free the code analyst from routine work while allowing concentrating at more complex tasks.

Working with these tools, however, presents its own set of issues. Producing the full set of code checking rules for the project requires significant effort. Predefined sets of rules packaged with tools seldom satisfy the developer; in these cases custom rules need to be created. If such rules have to be implemented using general-purpose languages such as Java or C# [5, 6, 13], this becomes time-consuming and routine work: it is necessary to provide a technical "boilerplate" code enabling the features provided by the tool (e.g. initializing its API), implement the rule-checking code itself, and create the set of configuration parameters allowing run-time rule customization. As a result, *rule reuse problem* becomes important. Unfortunately, reusing complete sets of rules is not an option in most cases as the projects and tools usually

J. Yang et al. (Eds.): UNISCON 2009, LNBIP 20, pp. 91–102, 2009.

differ enough to make such action impossible. On the other hand, it is desirable to perform at least partial rule reuse as the projects often require similar rules.

One of the problems with rule reuse is that currently the rationale of applying the rules in particular projects is often not documented explicitly. Rules appear "out of developer's experience" as no rule traceability back to the previous stages of the software process is supported. As a result, the reuse of such rules becomes difficult as nothing suggests the developer that the current project offers the prerequisites for the rules already used in some previous project.

In this paper, we will show how the rule reuse problem can be addressed via investigating the rationale behind the code checking rules. To do this, we turn to earlier steps of the software process (architectural design and requirements engineering) and look at design decisions, design patterns, and software quality requirements as possible candidates for this rationale. We argue that knowing the origin of every rule helps to achieve its reuse in future if the same context is encountered again during the development. To facilitate reuse, we propose establishing the rules repository where the rationale information is stored alongside the information about the rules themselves.

The rest of the paper is organized as follows. Section 2 presents the background information. Section 3 investigates possible rationale for application of the rules; section 4 presents a conceptual model of code checking rules and their rationale which can be used as a foundation of rules repository schema. Section 5 outlines possible reuse cases and scenarios using the knowledge of the code checking rules rationale. Section 6 surveys the related work, following by a conclusion and future work description in Section 7.

2 Background

In this section, we briefly introduce code checking rules and some candidates for their rationale, in particular, design decisions and quality requirements.

2.1 Code Checking Rules

The main purpose of static code analysis tools [29, 31] is establishing the set of code checking rules looking for common bugs and bad practices in software and reporting the results to the analyst. Such rules define undesirable or prohibited code solutions (code anti-patterns) usually expressed in terms of the constructs of the particular target programming language (class, interface, method, property etc.) and other applicable (e.g. API-related) concepts. Such solutions include e.g. invalid usage of a particular class, undesirable sequence of method calls, improper access to a particular method from other particular method, violating some guidelines for class implementation or API usage (e.g. failing to implement a mandatory interface or property).

Let us look at an example of a code checking rule. Maintaining dependencies between the components in multi-tier architecture may lead to the restriction of using particular set of classes from other particular set of classes, for example, presentation tier classes cannot be used from business logic classes. This leads to the following rule: *it is forbidden to use types belonging to a presentation tier from any types belonging to*

a business logic tier. An example of an implementation of this rule (using Code Query Language (CQL) [3] supported by NDepend tool) is as follows:

```
WARN IF Count > 0 IN SELECT METHODS FROM "BusinessLogic"
WHERE IsDirectlyUsing "PresentationLogic"
```

2.2 Design Decisions and Quality Requirements

Design decisions are "the significant decisions about the organization of a software system" [22]. Such decisions are made during the design step of the software process and reflect the view of the system that needs to be implemented. Actually, every design decision directly controls the code development. The extensive ontology of such decisions is proposed by Kruchten [23], it identifies three major decision categories: existence decisions (related to some functionality present or absent in a system), property decisions (related to some quality attribute supported by a system) and executive decisions (not related directly to the functional elements or their qualities). In recent works, the complete software architecture tends to be viewed as a set of such decisions [20]. Examples of design decisions can be "the system will have three-tier architecture", "the system will include the cache manager using up to 5 background threads and allowing up to 1000 concurrent clients".

Software quality requirements represent the desirable quality characteristics for a system [9, 12] (such as availability, performance, safety, or security) as seen by its stakeholders. Examples of quality requirements are "response time for operations with a customer order must not exceed 0.2 sec under the load up to 700 requests per second", "all attempts to have access to the order processing system must be authorized". Such requirements are actually supported or opposed by corresponding design decisions [35], we will look at this issue in detail in subsection 3.3.

3 The Rationale Behind the Code Checking Rules

In this section, we show how the rationale behind the code checking rules can be revealed. We trace this rationale first to existing design decisions and then to corresponding software quality requirements.

3.1 Code Checking Rules and Design Decisions

To find the rationale behind the application of a particular code checking rule, we need to look at the activities belonging to earlier steps of the software process. We argue that *as the set of code checking rules enforces a particular design decision, it is natural to accept that decision as a direct rationale for these rules*.

For example, the design decision "the logical view is organized in 3 tiers named DataAccessLogic, BusinessLogic and PresentationLogic" is a direct rationale for establishing the following code checking rule "no methods of classes belonging to PresentationLogic package can be called from either DataAccessLogic or BusinessLogic packages" as the creation of this rule is a direct consequence of accepting the design decision.

As a software architecture can be seen as a set of interconnected design decisions [20] and Kruchten's design decision ontology [23] adds to this set the decisions not related to architecture (introducing the executive decisions related to organizational issues, the choice of technology or a development tool) we argue that *every code checking rule can be traced to the corresponding design decision*. Actually, such rule is an integral part of the implementation of this design decision.

The design decision usually contains some information that can be used to parameterize the corresponding rules (Fig.1). For example, if it was defined that the presentation layer code is contained in a package named *PresentationLogic*, the name of the package parameterizes the rule restricting an access to this package. Numeric information (such as cache size, maximum number of threads etc) can be used this way as well. Reusing such rules requires defining the values for specified parameters.

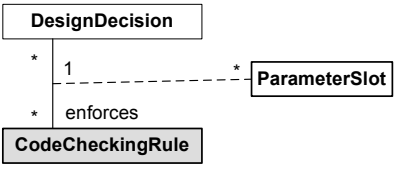

Fig. 1. Design Decision and Code Checking Rule

Specifications of design decisions can be treated to some degree as *code quality requirement specifications* as they define the desired quality properties of the corresponding implementation code. These specifications are also similar to requirement specifications as they are often informal (actually, the design decisions are often left expressed in natural language). In fact, some kind of NLP-based requirement elicitation and analysis can be performed to actually map these specifications into code checking rules (similarly to what is proposed by conceptual predesign [26] and the NIBA project [10] to map traditional software requirements into design-time notions). This mapping is a target for future research, in this paper we treat the design decisions as "black boxes" assuming they are specified in a format that allows retrieving the information intended for rule parameterization.

3.2 Code Checking Rules and Design Patterns

From the point of view of facilitating reuse, it would be even more interesting and useful to look at *design patterns* as possible rationales for code checking rules. The simplest way to deal with this case is to treat the pattern as a special kind of design decision which has predefined meaning and can be applied in different contexts (this is the view accepted in e.g. [20]). In this case, the set of code checking rules can be seen as enforcing the design pattern the same way as it enforces any other design decision. Even in this case, however, we would recommend distinguishing design patterns from other (ad-hoc) design decisions as it would greatly help in documenting the likely chances of reuse (as design patterns in most cases become reused more often than ad-hoc design decisions).

Another approach is to view design patterns as "enablers" for several design decisions at once (as original GoF book states "once you know the pattern, a lot of design decisions follow automatically" [11]). In this situation, a design pattern can be treated as a rationale for all code checking rules corresponding to the enabled decisions. For example, applying *Model-View-Controller* design pattern enables the set of existence design decisions related to defining the program units implementing all parts of a solution (such as "the package *DataView* will contain the code for the view class and all supporting program artifacts"). These decisions, in turn, are enforced by a set of code checking rules (e.g. prohibiting direct access from the model to the view).

Making able to relate code checking rules to design patterns is probably the most important reuse facilitation technique as such rules become reused frequently.

3.3 Code Checking Rules and Quality Requirements

Now we can investigate the possibility of tracing the rationale of code checking rules further – to *software quality requirements* (those defined by the stakeholders in a process of requirements elicitation). Being able to relate static code checking rules to software quality requirements allows for better understanding the interdependencies between external and internal software quality (as quality requirements represent external quality and code checking rules are supposed to enforce internal quality).

To understand the relationships between quality requirements and code checking rules, first look at the relationships between quality requirements and design decisions [35]. These relationships are depicted on Fig.2. They are of "many-to-many" kind as achieving the particular quality requirement can be affected by several design decisions (e.g. the desired performance can be achieved with a help of creating a cache with a specific size and going multithread) whereas a particular design decision can affect achieving several quality requirements (e.g. the cache of the particular size can help both performance and scalability). Also it is necessary to introduce the degrees of affection as it can be either positive or negative. Some decisions can actually break quality requirements as it is not always possible to satisfy all such requirements due to implementation constraints and some compromise need to be found ([2] uses the term "satisficing" referring to such requirements). For example, the design decision "a cache database allowing up to 500 simultaneous users will be used by a cache manager" can support the quality requirement "a response time under the load up to 500 users must not exceed 0.2 sec" while breaking the requirement "a response time under the load up to 1500 users must not exceed 0.5 sec".

Now it is possible to trace the code checking rules to the software quality requirements. As the specific quality requirement is affected by the specific design decision either positively or negatively, it can be also affected the same way by all code checking rules that enforce this design decision. So actually we have several groups of code checking rules related to particular quality requirement – one group per degree of possible affection. In this paper we restrict ourselves to only two groups: *supporting rules* and *breaking rules*. It is important to understand, however, that if the particular rule is listed for the particular requirement as "breaking" it does not necessarily mean that its

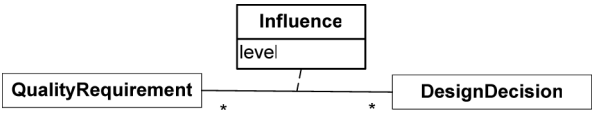

Fig. 2. Quality Requirement and Design Decision

implementation and enforcement necessary breaks this requirement: it only means that it enforces breaking design decision so it has more chances to be harmful and needs to be investigated more carefully.

Some code checking rules cannot be traced to quality requirements, in particular, the rules driven by business environment i.e. related to accepted coding standards, tools to be used etc. (corresponding to the executive decisions according to Kruchten's ontology [23]). This fact has to be captured in a way that makes it able for an analyst to locate all the rules of this kind.

3.4 Code Checking Rules and Software Quality Characteristics

Software quality requirements refer to specific quality characteristics belonging to particular quality model (such as e.g. defined by ISO/IEC 9126 quality model standard [18]). These models are usually defined as taxonomies of quality characteristics with specific software metrics belonging to the lowest level. For example performance requirement "The response time for order processing must not exceed 0.5 sec" (*R1*) refers to "response time" quality metric which belongs to "time behavior" middle-level characteristic and "efficiency" upper-level characteristic according to ISO/IEC 9126 quality model specification.

As we have already connected code checking rules to software quality requirements, we can see that this connection contains information related to association between the rules and corresponding quality characteristics. For example, all the rules related to *R1* requirement can be associated with a "response time" metric and two upper-level quality characteristics: "time behavior" and "efficiency".

Connecting code checking rules to quality characteristics can be seen as rule categorization, it does not involve parameterization of the rules (as in case of design decisions) and does not include the degree of affection (there are no "rules breaking response time" as "response time" in this case is only a category label). If such parameterization is undesirable, the analyst can even skip the detailed description of the relationship between the rule and the design decision and instead only record the connection between the rule and the corresponding quality characteristic.

Another quality-based code checking rule classification is proposed in [27]. Using the tool proposed in that paper, it is possible to specify the code quality characteristic and obtain all the rules associated with it. This approach, however, is restricted to code quality characteristics that characterize internal software quality and can be directly associated with the rules. Our approach allows revealing the indirect connection between code checking rules and external software quality characteristics. These two classifications are actually complimentary and can be used alongside each other.

4 A Conceptual Model of Code Checking Rules and Their Rationale

After achieving understanding of the relations between the rules, design decisions, and quality requirements we can describe how this information can be used by the tool aimed at facilitating rule reuse. We propose to establish the rules repository where the information about the available rules together with their rationale is stored and make the tool access this repository. The conceptual model of code checking rules making use of these relations can be used to establish a schema for this repository. This model is shown on Fig.3; for brevity, we omitted most attributes and the software project-related part of the model.

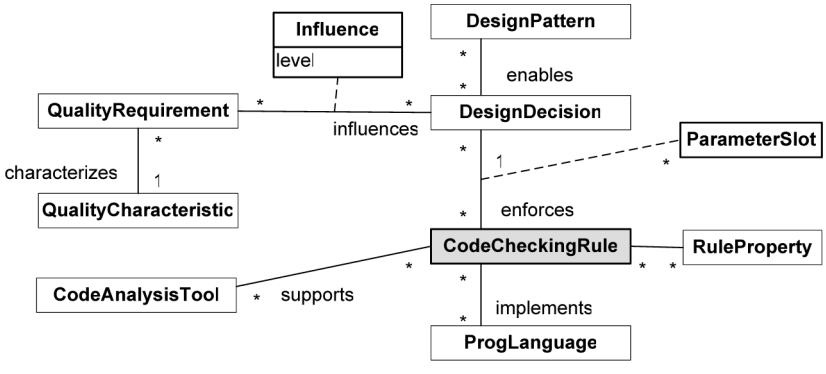

Fig. 3. A Conceptual Model of Code Checking Rules and Their Rationale

In this model, the relationships between the rules and their properties are of "many-to-many" kind. It reveals the fact that we treat these properties as "tags"; as a result, every rule can have sets of tags of different kind associated with it. This tagging approach allows for additional flexibility of expressing the rules; the repository based on this model is supposed to offer better search capabilities.

Design decisions are also treated as tags as there can be many design decisions connected to the particular code checking rule (probably originated from different projects). The model allows the analyst to drill down the particular design decision to see all the corresponding quality requirements, quality characteristics etc.

The model reflects the fact that not every programming language or static analysis tool offer the support for the particular rule (see [34] for more formal treatment of this issue). Some rules are specific for a particular language or framework (e.g. namespace-related rules for .NET); some tools, on the other hand, do not offer capabilities to implement specific categories of rules. To deal with this fact, the information about the available language and tool support is stored with the rule. If the particular support is missing, the analyst attempting to locate the rule in a repository should receive the meaningful explanation of the problem. This approach also have an advantage of being flexible enough to incorporate future events when new language constructs are

introduced to match the capabilities of other languages (e.g. Java generics) or the tools start to support new features.

The model offers limited support for rule parameterization; actually it allows defining the set of parameter slots to transfer the information between the design decision specification and the code checking rule (both treated as "black boxes" as stated earlier). How this information can be used to actually parameterize the rule is left to the implementation of the rule reuse support tool. In future, we plan to make use of language- and tool-independent metamodel (similar to what is proposed in [34]) to describe the rule functionality, in this case, we will precisely define the support for rule parameterization (treating the rule as a "white box").

5 Rule Reuse

In this section, we look how the proposed notions of code checking rule rationale can be used in practice to facilitate reuse. We start from the common list of reuse cases and later introduce two possible rule reuse scenarios: top-down and bottom-up.

5.1 Rule Reuse Cases

Our model facilitates the reuse of the code checking rules for the following cases (suppose we have established the special repository where the information about the available rules and their rationale is registered; the structure of this information follows our conceptual model):

1. Implementing the registered design decision: it is possible to reuse all the corresponding rules directly with all the parameter values (rare case of "copy-paste" reuse), perform the corresponding parameterization, reject some rules, or use some combination of these actions.
2. Applying the registered design pattern: the most frequent action is to reuse all the corresponding rules with parameterization.
3. Encountering the registered quality requirement: it is possible to consider reusing the corresponding supporting rules (the actual reuse will be probably postponed until the actual design decisions are defined).
4. Investigating the consequences of supporting the particular registered quality characteristic: all the code checking rules corresponding to this characteristic can be considered for reuse and investigated in detail.
5. Working on a project which is similar to already-performed project: all the design decisions of the past project can be examined together with their corresponding code checking rules.

5.2 Rule Reuse Scenarios

Let us look at two typical scenarios for the reuse of the code checking rules: bottom-up and top-down.

Top-down scenario is a straightforward one:

1. The analyst implements the design decision to support the particular quality requirement. This decision has to be enforced with a set of code checking rules.

2. The search for a quality requirement in a repository is performed. If it is not found new requirement is registered, otherwise case 3 (see subsection 5.1) is followed;
3. The search for a design decision in a repository is performed. If it is not found new decision is registered in a repository, otherwise case 1 is followed;
4. If on any of the previous stages the corresponding checking rules cannot be found, they should be created and registered as well.

Bottom-up scenario should occur less frequently and is more difficult to follow:

1. The analyst works with some static code analysis tool on a particular application (it can be an existing application being refactored etc.)
2. The set of errors and imperfect code fragments is found; some of them are believed to have recurring character.
3. The synthesis of the code checking rules is performed for these errors; the information about these rules is registered in a repository.
4. These rules are associated to the possible design decisions that could serve as their rationale. This activity can involve non-trivial reasoning (in reverse engineering sense) if design decisions have yet to be defined. In complicated cases, only the categorization of the rules according to quality characteristics can be performed (this is usually easier to do).
5. If quality requirements specification is defined for a system, design decisions can be related to quality requirements.

Described reuse cases and scenarios can serve as parts of requirements specifications for a reuse support tool with underlying code checking rules repository. This tool is currently under development.

6 Related Work

We can classify the related work on rule reuse into two categories: (1) low-level approaches addressing rule reuse on the code level ("white-box" reuse) and (2) higher-level approaches investigating the place of such rules in a software process, their relations to the goals of the project, its quality requirements etc.; these approaches treat the rules as "black boxes" to some degree (with possible parameterization). Most of the work in this area falls into the first category. We found very few works investigating the application rationale of the rules.

6.1 Source-Level (White-Box) Rule Reuse

Custom code analysis rules have yet to be extensively investigated by the research community, most of the publications related to such rules are "how-to" (mostly online) guides targeting particular industrial [5-7, 13] or research [16] tools; few attempts to achieve reuse across tool or language boundaries are made. As a result, the proposed solutions are actually rather low-level; they concentrate on specifics of rule implementation for particular tools. An exception is a paper [7] where in addition to targeting a Fortify tool an attempt is made to express the rules using high level (language- and tool-agnostic) description language with defined grammar; no grammar, however, is actually introduced in a paper.

A common white-box approach for rule reuse is to start from building a conceptual rule model and then instantiate this model in different contexts. Jackson and Rinard in their roadmap for software analysis [19] emphasized the importance of this approach. They suggested using special generic code representation as a model and build all the code analysis activities on top of this model. The list of code models suitable to be used in model-driven software analysis is rather extensive; it includes the schemas for XML-based source code exchange formats such as GXL [15], srcML [4, 24], or OOML [25], semantic code models such as ASG [8]. Detailed discussion related to interrelationships between code models and code exchange formats is presented in [21]. The specific case of using and adapting srcML for the support of code analysis is discussed in [24].

An extensible meta-model-based framework to support the code analysis is described in [34]. This approach provides the most extensive white-box reuse capabilities. It formally defines the relationships between generic representation of the source code, conditions specified on top of this representation (which can represent code analysis rules), and specifics of particular languages and tools (front-ends for the common metamodel). The definition of the common meta-model underlying the framework reflects these issues, rules expressed via this metamodel can be easily reused across tool and language boundaries. The metamodel, however, does not provide any means to aid capturing design-time rationale of rule application.

6.2 Code Quality and Black-Box Rule Reuse

An approach utilizing code quality models to assess open-source software projects using extensive set of metrics was developed as a result of an SQO-OSS project [30]. Other approaches aimed at this goal are presented in [28, 32]. Such approaches are limited to integrated quality assessment (i.e. they produce the values for software quality attributes and an integrated value for overall product quality), they do not address rules necessary to enforce quality. Their quality characteristics, however, can be useful in reuse contexts as the rules can be associated with such characteristics using an approach shown in subsection 3.4.

Connecting code checking rules to a quality model was proposed in [14, 27]; we briefly discussed this work in the subsection 3.4. This connection is introduced as a part of specific code analysis process called EMISQ. Both quality model and an evaluation process are based on ISO 14598 and ISO 9126 standards. No model for internal structure of the rules is proposed, instead, the available rules are stored into the repository after classifying according to (1) code quality characteristics they help to enforce and (2) available implementations (language and static analysis tool support). The proposed tool aids the developer allowing fast filtering of the available set of rules according to a quality characteristic they are supposed to enforce, required tool or language support etc. The EMISQ approach is close to our technique with respect to admitting the need for capturing the connections from the rules back to the system goals (which can be seen [2] as corresponding to software quality requirements). Actual implementation of this connection, however, is not presented in a paper and its use to facilitate rule reuse is not investigated. Also this approach does not take into account design decisions and patterns. In fact, to aid rule reuse, the developer is supposed to pick context-relevant code quality characteristic manually and choose the appropriate rules without any specific connection to the past development experience.

7 Conclusions and Future Work

In this paper, we proposed an approach for revealing the rationale behind the application of code checking rules in static code analysis. We stated that these rules can be related to design decisions and patterns, quality requirements and quality characteristics. We presented a conceptual model for the code checking rules and their rationale; this model can be used to establish a schema for the rule repository underlying reuse support tools. The proposed solution facilitates rule reuse in recurring development contexts by allowing the analyst to look at all the rules related to the particular context.

In future, we plan to implement a tool support for our approach. Prospective tool will offer a repository for code checking rules organized in correspondence with the proposed conceptual model allowing analysts to perform queries for available rules according to different reuse cases and scenarios. Another direction of research is transition to the "white box" representations of the code checking rules (according to [34]) and the design decisions (e.g. according to their UML profile [35]).

References

1. Chess, B., West, J.: Secure Programming with Static Analysis. Addison-Wesley, Reading (2007)
2. Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers, Dordrecht (1999)
3. Code Query Language 1.8 Specification (accessed January 11, 2008), http://www.ndepend.com/CQL.htm
4. Collard, M.L., Maletic, J.I., Marcus, A.: Supporting Document and Data Views of Source Code. In: Proc. DocEng 2002. ACM Press, New York (2002)
5. Copeland, T.: Custom PMD Rules. OnJava.com (2003) (accessed January 11, 2008), http://www.onjava.com/pub/a/onjava/2003/04/09/pmd_rules.html
6. Create Custom FxCop Rules (accessed January 11, 2008), http://www.thescarms.com/dotnet/fxcop1.aspx
7. Dalci, E., Steven, J.: A Framework for Creating Custom Rules for Static Analysis Tools. In: Proc. Static Analysis Summit, pp. 49–54. Information Technology Laboratory, NIST (2006)
8. DATRIX Abstract Semantic Graph Reference Manual, version 1.4. Bell Canada (2000)
9. Firesmith, D.: Using Quality Models to Engineer Quality Requirements. Journal of Object Technology 2, 67–75 (2003)
10. Fliedl, G., Kop, C., Mayerthaler, W., Mayr, H.C., Winkler, C.: The NIBA Approach to Quantity Settings and Conceptual Predesign. In: Proc. NLDB 2001. LNI, vol. P-3, pp. 211–214. GI (2002)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns. Elements of reusable object-oriented software. Addison-Wesley, Reading (1995)
12. Glinz, M.: Rethinking the Notion of Non-Functional Requirements. In: Proc. Third World Congress for Software Quality (3WCSQ 2005), Munich, vol. II, pp. 55–64 (2005)
13. Grindstaff, C.: FindBugs, Part 2: Writing custom detectors. IBM Developer Works (2004) (accessed January 11, 2008), http://www.ibm.com/developerworks/library/j-findbug2

14. Gruber, H., Körner, C., Plösch, R., Schiffer, S.: Tool Support for ISO 14598 based code quality assessments. In: Proc. QUATIC 2007. IEEE CS Press, Los Alamitos (2007)
15. Holt, R.C., Winter, A., Schürr, A.: GXL: Toward a Standard Exchange Format. In: Proc. WCRE 2000, pp. 162–171 (2000)
16. Holzmann, G.J.: Static Source Code Checking for User-Defined Properties. In: Proc. IDPT 2002. Society for Design and Process Science (2002)
17. IEEE Standard for Software Reviews. IEEE Std 1028-1997. IEEE (1997)
18. ISO/IEC 9126-1, Software Engineering – Product Quality – Part 1:Quality model. ISO (2001)
19. Jackson, D., Rinard, M.: Software Analysis: A Roadmap. In: Proc. Conf. on The future of Software engineering. ACM Press, New York (2000)
20. Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: Proc. WICSA 2005, pp. 109–120. IEEE CS Press, Los Alamitos (2005)
21. Jin, D.: Exchange of software representations among reverse engineering tools. Technical Report. Department of Computing and Information Science, Queen's University, Kingston, Canada (2001)
22. Kruchten, P.: The Rational Unified Process - An Introduction. Addison-Wesley, Reading (1995)
23. Kruchten, P.: An Ontology of Architectural Design Decisions in Software-Intensive Systems. In: 2nd Groningen Workshop on Software Variability Management (2004)
24. Maletic, J.I., Collard, M.L., Kagdi, H.: Leveraging XML Technologies in Developing Program Analysis Tools. In: Proc. ACSE 2004, pp. 80–85. The IEE Publishers (2004)
25. Mamas, E., Kontogiannis, K.: Towards Portable Source Code Representations Using XML. In: Proc. WCRE 2000, pp. 172–182. IEEE CS Press, Los Alamitos (2000)
26. Mayr, H.C., Kop, C.: Conceptual Predesign - Bridging the Gap between Requirements and Conceptual Design. In: Proc. ICRE 1998, pp. 90–100. IEEE CS Press, Los Alamitos (1998)
27. Plösch, R., Gruber, H., Hentschel, A., Körner, C., Pomberger, G., Schiffer, S., Saft, M., Storck, S.: The EMISQ Method - Expert Based Evaluation of Internal Software Quality. In: Proc. 3rd IEEE Systems and Software Week. IEEE CS Press, Los Alamitos (2007)
28. Rentrop, J.: Software Metrics as Benchmarks for Source Code Quality of Software Systems. Vrije Universiteit, Amsterdam (2006)
29. Rutar, N., Almazan, C.B., Foster, J.S.: A Comparison of Bug Finding Tools for Java. In: Proc. ISSRE 2004, pp. 245–256. IEEE CS Press, Los Alamitos (2004)
30. Samoladas, I., Gousios, G., Spinellis, D., Stamelos, I.: The SQO-OSS quality model: measurement based open source software evaluation. In: Proc. OSS 2008, pp. 237–248 (2008)
31. Spinellis, D.: Bug Busters. IEEE Software 23, 92–93 (2006)
32. Stamelos, I., Angelis, L., Oikonomou, A., Bleris, G.L.: Code quality analysis in open source software development. Info. Systems J. 12, 43–60 (2002)
33. Stellman, A., Greene, J.: Applied Software Project Management. O'Reilly, Sebastopol (2005)
34. Strein, D., Lincke, R., Lundberg, J., Löwe, W.: An Extensible Meta-Model for Program Analysis. IEEE Transactions on Software Engineering 33, 592–607 (2007)
35. Zhu, L., Gorton, I.: UML Profiles for Design Decisions and Non-Functional Requirements. In: Proc. SHARK 2007. IEEE CS Press, Los Alamitos (2007)