Using Automated Tests and Restructuring Methods for an Agile Development of Diagnostic Knowledge Systems

Joachim Baumeister and **Dietmar Seipel,** and **Frank Puppe**
Department of Computer Science, University of Wuerzburg, Germany
email: {baumeister, seipel, puppe}@informatik.uni-wuerzburg.de

Abstract

We introduce an agile process model for the development of diagnostic knowledge systems. As key practices of this evolutionary process model we identify structured modifications of the knowledge by restructuring methods, and the continuous validation of the knowledge using automated testing methods. Finally, we report promising experiences gained by the evaluation of a development project implementing a medical documentation and consultation system.

Introduction

Classical process models are often not appropriate for developing diagnostic knowledge systems, if, e.g., a full specification is not known beforehand, and the project team is small. In our context, we investigated the development of medical knowledge systems, that were typically constructed and maintained by 1-2 physicians. In that case, the knowledge is usually formulated by the domain specialists themselves. If we consider classical process models like, e.g., CommonKADS (Schreiber *et al.* 2001), then we see that they are facing some problems, when applied under the circumstances described above:

- Technical feasibility is not always known in advance. A small prototype would provide a good basis for deciding about the overall project.
- Full specification is often not known in advance. During the development of a knowledge system new requirements arise and previously defined requirements become less important.

Moreover, for small-size or mid-size development projects document-centred approaches seem to be cumbersome and costly for domain experts. Verbose specifications and necessary decisions about the project design often deter experts from starting or continuing a knowledge system project. In fact, experts were motivated, if early results would be taken out from small specifications, and, if systems could grow incrementally from a small pilot.

In this paper, we briefly introduce a novel process model, which allows for an agile construction and maintenance of diagnostic knowledge systems. We identify the automated validation and restructuring of knowledge as the key practices for a successful application of the agile process model.

In knowledge engineering research, validation techniques have been undergoing fruitful research for the last decades. Classical work by (Coenen & Bench-Capon 1993) was continued, e.g., by (Preece 1998). Especially, for the rule-based implementation of knowledge systems an extensive framework was defined by (Knauf 2000). Interestingly, this framework does not only describe the evaluation of test knowledge (represented as test cases), but especially its suitable generation. In the context of our projects, test cases were either already available or manually defined by the experts, which on one hand raises the question for completeness, and on the other hand is a complex and time consuming task. For this reason, the integration of a formal method generating suitable test cases will be a promising extension of the presented work.

The introduction of restructuring methods for a step-wise and algorithmic modification of the knowledge base was inspired by refactoring methods introduced for software engineering (Opdyke 1992; Fowler 1999). As an advantage to general software engineering, where test case adaptation is mainly done manually, the implementation of restructuring methods for knowledge systems often can propagate their changes to the attached test knowledge, e.g., by modifying the corresponding objects in test cases. However, the refinement of the knowledge base performed by restructuring methods differs from refinement techniques, e.g., described by (Boswell & Craw 1999; Knauf *et al.* 2002), since restructuring is mainly *not* applied for improving the accuracy of the system, but for improving the design of the knowledge base, i.e., the structure of the implemented knowledge. Especially for this reason a restructuring method is currently initiated manually, though supported by automated adaptations of attached knowledge.

An interesting issue remaining partially solved is the maintenance of test knowledge. This problem was formulated by Menzies as the *recursive maintenance problem* (Menzies 1999). In his article, he argued that test knowledge was introduced to simplify the maintenance of knowledge, but the maintenance of test knowledge also needs to be considered sufficiently, possibly by meta-test knowledge. A partial solution for this problem will be presented by the automated propagation of the restructuring methods also adapting test cases according to the performed changes. Modifications of the knowledge base and the test knowledge with respect to a changing world, i.e., domain expertise, can be resolved by refinement techniques mentioned in the following.

The paper is organized as follows: The following section briefly introduces the agile process model and its steps, respectively. Based on this process model we describe automated tests and restructuring methods in the next sections. Thereafter, we report promising experiences gained by the evaluation of a development project implementing a medical documentation and consultation system. We conclude the paper with a summary of the presented work, and show possible directions for future research.

An Overview of the Agile Process Model

The presented process model was inspired by the agile process model eXtreme programming (XP). In software engineering research and practice XP (Beck 2000) has attracted great attention, and showed its significance in numerous projects. An agile process model has the following properties:

- an early, concrete and continuing feedback
- an incremental planning approach
- a flexible schedule of the development process
- the design process lasts as long as the system lasts

In contrast to XP the presented process model does not consider the implementation of general software using an all purpose programming language (e.g., C or Java), but focusses on the development of knowledge systems, that are constructed by the insertion, the change, and the review of knowledge represented in a declarative language.

The Steps of the Agile Process Model

The agile process model (cf. Figure 1) is a light-weight process model and consists of the following cycle: analysis of the system metaphor, design of the planning game, implementation (plan execution: including tests, restructuring and maintenance), and integration. We briefly discuss these 4 steps in the following.

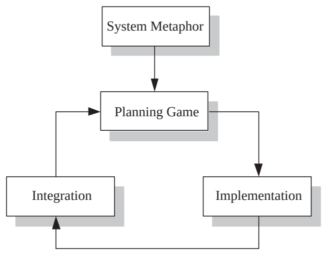

Figure 1: The agile process model for developing knowledge systems.

The System Metaphor. The system metaphor describes the basic idea and the designated goals of the knowledge system to be implemented, and it is used to facilitate a better communication between the developer (i.e., the domain specialist) and the user of the system. Thus, the metaphor stands for a common system of names and a common system description. Using a common system metaphor can greatly simplify the development and the communication between user and developer.

Thus, the *local system metaphor* defines names and semantics of the basic entities used during the development of the system, e.g., diagnoses (solutions), questions (input), and cases (solved problems). The *global system metaphor* describes the overall idea of the system to be implemented. Typical classes are the *consultation system* (focussing on the inference of solutions), the *documentation system* (focussing on the standardized and correct acquisition of input data, and the *embedded system* (focussing on the technical integration of the system into an existing machine).

The Planning Game. The planning game is the starting point of the development cycle: During the planning game the developer and the user decides about the scope and the priority of future development, i.e., extensions or modifications of the current system. For each extension/modification a plan is defined, which is documented by *story cards*. Besides the desired functionality the costs and priority of each plan are estimated. Based on the estimated values of these factors the developer and the user define the next release by selecting story cards. They define the scope of the release by ordering the collected cards and defining a release deadline according to the risk estimations made before. It is worth noticing, that these factors provide a benchmark for feedback in order to enable adaptation of plan estimation in the future.

The planning game provides a flexible method for guiding the development process of knowledge systems. On one side, plans are documented in story cards and deliver a structured sequence of the development process, in which the user as well as the developer are integrated. On the other side, during the definition of stories the user and the developer specify the expected behavior of the planned knowledge extension, and thus prepare useful validation knowledge for the subsequent implementation phase. Furthermore, by providing methods for estimating and documenting implementation costs (derived from the implementation velocity), an accurate feedback can be given to assess the whole development process.

The Implementation. The implementation phase considers the realization of the story cards specified in the planning game phase. In the context of the agile process model, the implementation phase follows a test-first approach: Any implementation of the functionality of a story is preceded by the implementation of appropriate tests. Therefore, we distinguish between a *test-implementation phase* and a *code-implementation phase*.

In the test-implementation phase the developer defines test knowledge describing the expected behavior of the new story to be implemented. The kind of test knowledge depends on the representational language of the code-implementation. It is easy to see that, e.g., the test knowledge for a rule-based knowledge representation can differ from the test knowledge of a model-based representation.

Test knowledge needs to be automatically executable, i.e., the results of the test can be evaluated automatically by the system. As a main idea of the process model, the continuous application of the cyclic process yields a suite of tests, which can be executed as a whole. We discuss the importance of automated tests and test suites in the following sections in more detail.
The code-implementation phase considers the actual realization of the story, e.g., by acquiring and formulating new knowledge, or by restructuring existing knowledge. In the context of this paper we omit a detailed description of this sub-phase.

The Integration. If the newly implemented functionality passes the corresponding tests and the test suite, respectively, then this knowledge is committed to the knowledge base. Since the integration is done continuously, we always can access a running system enclosing the currently implemented knowledge. For a reasonable integration additional tests need to be available, which are too time consuming to be included into the working test suite, but which are applied during integration to check more aspects of the functional behavior of the knowledge system. We call these tests *integration tests*. Integration tests often contain a larger number of previously solved cases, which can be run against the knowledge system. These previously solved cases typically contain a set of question-answer pairs and a set of expected diagnoses for these pairs, but sometimes also knowledge about dialog behavior is available. Running thousands of cases can take several minutes or hours. Therefore, it is not practical to include them into the working test suite, since the suite usually is applied many times during the implementation of a story. Nevertheless, before the integration of a new version these integration tests are an essential indicator for the correct behavior of the knowledge system.

Automated Tests

One of the key properties of the agile process model is the application of automated tests. For an automated application of tests the expected result of a test needs to be known beforehand. The most prominent example for automated tests is the use of empirical testing, i.e., running previously solved test cases. But there exist further validation techniques, which can be simply adopted for automation. There exist tests that could be executed without any test knowledge, e.g., anomaly testing. However, some tests necessarily require test knowledge, which has to be defined by the expert, e.g., empirical testing requires the presence of appropriate test cases.

Significance of Tests

At first sight the construction of tests is an additional and huge effort during the implementation phase. Nevertheless, implementing tests besides the actual functionality is good for the following reasons:

- *Validation of the code:* Tests are primarily defined to validate the subsequent implementation. If the system passes the tests, then the developer and the user feel confident, that the newly implemented functionality has the expected behavior.
- *Removing communication errors:* Tests are often implemented as examples of typical system runs. Defining such examples in cooperation with the user (which has to know the typical system behavior) will clarify story definitions. It is worth noticing, that often ambiguous definitions are timely exposed due to the test-implementation phase.
- *Detecting side effects:* Since all tests are collected in a common test suite, all available tests will be executed before completing the implementation of a story. Thus, side effects can easily be discovered, i.e., a new functionality has accidentally changed the behavior of a previously implemented functionality.

For this reason, we identify testing as one of the key practices of the agile process model. In the following, we classify tests methods with respect to their validated target and we describe the application of automated tests.

Classification of Tests

In general, we can classify test methods according to the properties of the knowledge that should be validated. Thus, we identify methods for validating the *correctness* of the knowledge system, for finding *anomalies* contained in the system, for testing the *robustness* of the system, and for testing the *understandability* of the implemented knowledge. Methods for these types are briefly discussed in the following.

Correctness. In the past, testing has been focussed on validating the correctness of a system. For example, empirical testing is the most popular method for correctness testing. Here, the implemented knowledge system runs previously solved test cases and infers solutions for each case. The inferred solutions are compared with the stored solutions of the case and differences are presented to the user. If a sound dialog behavior of the system is also of interest, then *sequentialized test cases* can be used, i.e., ordinary test cases augmented with the correct sequence of asked questions.

Anomalies. The detection of anomalies is also an important issue in validation research. An anomaly is defined as a certain part of the knowledge base, which a priori is not incorrect but *can* cause the system to behave irregularly. Examples for anomalies are redundant, cyclic, or ambivalent knowledge. For rule-based systems (Preece, Shinghal, & Batarekh 1992) have presented a classification of anomalies and methods for detecting them. In principle, the classification of anomalies can be seamlessly transferred to other representations, e.g., case-based reasoning and Bayesian networks.

Robustness. If the knowledge system is intended to be used in stressful environments, then the robustness of the system is an important issue to consider. Methods for testing the robustness have been presented by (Groot, van Harmelen, & ten Teije 2000). The robustness of a rule-based system is measured by degrading the quality of system input, i.e., applying noise to test cases, and by reducing the quality of the implemented structural knowledge, i.e., removing or slightly modifying implemented rules.

Understandability. The understandability of the implemented knowledge was only studied a little in the past. However, for the agile development of knowledge systems, the understandability of the working knowledge base is very important. Understandability of knowledge can be approximately measured by metrics, that take the (relative) size and complexity of knowledge into account. For example, the complexity of a rule base can be determined by the following factors: The overall number of rules, the average complexity of rules with respect to their conditions, the average number of rules per diagnosis, the average number of findings per diagnosis, that are applied in the corresponding rules. An extensive framework for determining rule base complexities are presented in (Atzmueller, Baumeister, & Puppe 2004).

For some of these methods additional test knowledge is required, e.g., empirical testing and robustness testing require appropriate test cases. Other methods can be executed even without any supplementary test knowledge, e.g., anomaly testing. It is worth noticing, that test knowledge is mostly acquired manually by the domain specialists during the implementation phase.

Automated Application of Tests

For the agile development of knowledge systems test methods need to be adapted to be executed automatically. Thus, the expected result of the particular tests need to be known beforehand. If any of the tests fail, then an error is reported. Otherwise, the successful pass of the tests is reported with a short success message.

In its simplest form, empirical testing can be automated without any adaptation, since previously solved cases are used for which the correct solution is already known. However, if the cases were gathered by a real life application, then we cannot expect all cases to be solved correctly. Then, an expectation value is required, which specifies the minimum percentage of cases to be solved correctly.

For finding anomalies the developer has to decide about the kind of anomalies, for which an error should be reported, e.g., ambivalency and circularity may indicate serious deficiencies of the knowledge base.

An expectation value also needs to be specified, if the robustness of a system should be tested. Thus, a threshold value ξ has to be set, which defines the maximum noisiness for which the knowledge system has to behave correctly. For example, if threshold $\xi = 0.1$, then the knowledge system should derive the correct solution for each case even if at least 10% of the original input data is noisy.

Methods for testing the understandability of the implemented knowledge heavily depend on expectation values. For example, diagnoses with exceptionally large derivation rules can point to parts in the knowledge base, that may require restructuring in the future. A smaller size of derivation knowledge for single diagnoses mostly increases the understandability of the rule base. For this reason, an expectation value needs to be defined, which specifies the upper bound of the term ”exceptionally large”.

Restructuring Methods

The agile process model proposes the evolutionary development of knowledge systems. Thus, a system grows by incremental extensions of new knowledge or by structured modifications of already implemented knowledge. We call such modifications of knowledge *restructuring methods*, and they are commonly used for refining the design of the knowledge base, which for example can be measured by understandability metrics. Since an evolutionary development process implies the application of modifications from time to time, we identify restructuring methods as the second key practice of the agile process model.

Restructuring methods for knowledge bases were inspired by refactoring methods known in software engineering research, e.g., (Fowler 1999). A restructuring method is described by five parts: A meaningful and unique *name*, a *summary* giving a brief description of the method, a *motivation* for using the method, a list of *consequences* reporting conflicts and restrictions of the method, and the *mechanics*, an algorithmic description of the accomplishment of the method.

Usually, a restructuring method is initiated by the decision of the developer. Common restructurings are the modification of the question type, e.g., a multiple-choice question included in the knowledge base should be changed to a set of semantically equal yes/no questions. We call this method TRANSFORMMC2YN. For example, in a medical domain the multiple choice question *”examination doppler”* with possible values $\{$*stenosis, insufficiency*$\}$ can be translated into the two yes/no questions depicted in Figure 2.

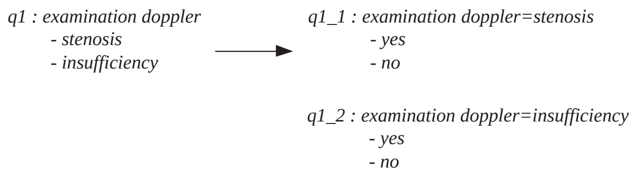
q1 : examination doppler
- stenosis
- insufficiency
q1_1 : examination doppler=stenosis
- yes
- no
q1_2 : examination doppler=insufficiency
- yes
- no

Figure 2: Restructuring of a multiple-choice question.

It is easy to see, that such modifications imply a subsequent change of already implemented knowledge, e.g., for TRANSFORMMC2YN we need to consider the change of all rules containing the original multiple-choice question. Another important issue to be considered is the coherent adaptation of test knowledge. For example, test cases need to be adapted according to the transformation, i.e., by exchanging the multiple-choice values with the corresponding yes/no answers.

Restructuring methods differ from ordinary knowledge modifications by specifying a structured procedure for implementing the particular modification. Thus, for each restructuring the mechanics and consequences of this modification are identified and described, e.g., the required adaptation of rules or test knowledge. An important implication can be drawn from this property: If the explicit procedure is defined for each restructuring, then the execution of such a method can be supported by interactive tools, that perform the required adaptations. This automatization of knowledge modifications enable developers to manage even complex changes of a knowledge base, that were very difficult to perform manually in the past. For example, the restructuring described above can imply the change of hundreds of rules and thousands of test cases, if applied to a real world knowledge base.

However, performing a restructuring can cause a knowledge base to become invalid. For example, a restructuring method decreasing the value range of an one-choice question can produce contradictory rules, if two rules have the same (transformed) one-choice answer in their rule conditions but contradictory rule actions. Therefore, any execution of a restructuring method is preceded by a *feasibility test*, which determines if the method will produce conflicts when executed. If this feasibility test fails, then the developer either uses default values to resolve the conflict or has to manually modify the knowledge base in order to remove the detected conflicts. Basically, the execution of a restructuring method passes the following tasks:

1 *Testing the actual state:* Before a restructuring method is executed the test suite is applied. A restructuring should only be considered, if the knowledge base is in a valid state.

2 *Feasibility test:* Checks, if the restructuring causes unresolvable conflicts, if executed on the existing knowledge base.

3 *Method application:* For all included knowledge (e.g., ontological objects, rules, models, cases) the following subtasks are performed:

   3.1 *Conflict resolution:* If the execution of the method causes conflicts, which are not resolvable by default values, then the conflicts are solved with interaction of the user. Otherwise, default values are used for conflict resolution.

   3.2 *Method execution:* If no conflicts are remaining, then the method is executed according to the specified restructuring mechanics.

4 *Testing the resulting state:* After the restructuring method has been applied, the knowledge base again is tested using the test suite. A restructuring method only is successful, if the restructured knowledge base is in a valid state.

A collection of 18 typical restructurings, faced in real-world knowledge system development, is extensively described in (Baumeister 2004). In summary, we apply restructuring methods to improve knowledge base design. Typical restructurings are the modification of question types or the extraction of rule conditions. Restructuring methods differ from ordinary knowledge modifications by their explicitly described procedure. Thus, the methods can be automated and tool support can be offered.

Experiences

The presented agile process model was successfully applied in the ECHODOC[1] project (Lorenz *et al.* 2003), a medical documentation and consultation system supporting the anaesthesiologist when performing a transesophageal echocardiography examination (TEE). The system is guiding the physician during the TEE examination, i.e., presenting appropriate questions according to the current examination phase, and suggesting diagnoses derived from the examination process.

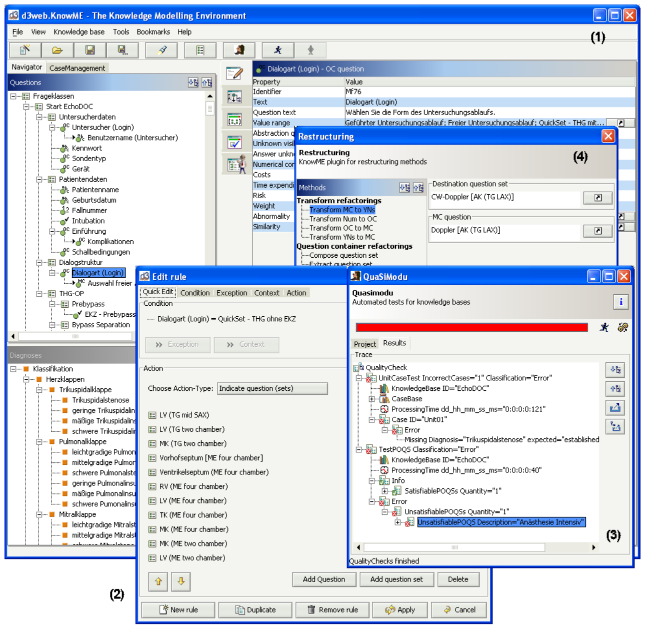

Figure 3: The knowledge modeling environment d3web.KnowME (1) with editors for defining rules (2), executing tests (3), and restructuring methods (4).

Currently, the knowledge base contains 252 questions grouped into 113 question sets, and 77 diagnoses. The rule base contains 254 rules implementing strategic knowledge defining the dialog behavior and structural knowledge for deriving solutions.

The domain specialists implemented the knowledge base using the knowledge modeling environment d3web.KnowME, which offers visual editors for the development of diagnostic knowledge systems. Furthermore, integrated editors for defining and executing automated tests and restructuring methods are included. Figure 3 depicts the main window of d3web.KnowME together with editors for editing rules, performing restructurings, and executing automated tests. Currently, the workbench provides editors for seven different test approaches (e.g., empirical testing for structural knowledge, sequentialized cases for testing the dialog behavior), and supports the automated execution of restructuring methods, that consider type transformations and hierarchical modifications.

First evaluations show, that the phases of the agile process model appeared to be very useful. Since the domain experts were only working part-time on the project, the use of stories in conjunction with the planning game was very advantageous. Due to the evolutionary nature of the development

[1]former QUALITEE

project the application of tests and restructuring methods appeared to be very significant. In most cases, modifications of the already implemented knowledge were performed successfully by using the restructuring methods; the methods commonly considered the change of question types or the rearrangement of the question hierarchy. It is worth noticing, that the rearrangement of the question hierarchy often implied the adaptation of the corresponding test cases representing the desired dialog behavior. Unfortunately, the offered restructuring methods only consider the corresponding inferential knowledge, but not the attached test knowledge, which has to be modified manually. However, it is planned to provide refined methods in the future providing an improved, automated behavior.

In summary, the domain specialists were more encouraged to perform (even bigger) changes on the knowledge base, because they were supported by appropriate restructuring methods, which are executing reproduceable changes on the knowledge. Additionally, test methods always ensured a valid state of the knowledge system, i.e., an expected behavior defined by the tests. A preliminary study of the practical application of ECHODOC has been undertaken, and a real-life evaluation of the system at the University of Wuerzburg Hospitals is currently planned and scheduled.

Conclusion

In this paper, we introduced an agile process model for developing (diagnostic) knowledge systems. We motivated that testing and restructuring are the key practices for a successful application of the evolutionary development of knowledge systems. Thus, we described the significance of tests for the development process in conjunction with important properties of test knowledge, e.g., their automated application. Further, we defined a scheme for classifying the different types of test methods. As the second key practice we described restructuring methods as a structured and algorithmic approach for modifying an existing knowledge base. Due to their explicit procedural definition restructuring methods can be automated, and tool support becomes possible.

The presented process model was evaluated on a running project developing a documentation and consultation system in the medical domain. During the development the significance of automated tests and restructuring methods was shown. First experiences are very promising, though there is still room for improvement, e.g., not all restructuring methods offer a propagation to the corresponding test knowledge. Finally, we think that also the inclusion of methods for automatic test case generation will be a very promising direction for future work.

References

Atzmueller, M.; Baumeister, J.; and Puppe, F. 2004. Quality Measures for Semi-Automatic Learning of Simple Diagnostic Rule Bases. In *Proceedings of the 15th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2004)*.

Baumeister, J. 2004. *Agile Development of Diagnostic Knowledge Systems (submitted)*. Ph.D. Dissertation, University Würzburg, Germany.

Beck, K. 2000. *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

Boswell, R., and Craw, S. 1999. *Organising Knowledge Refinement Operators, In: Validation and Verification of Knowledge Based Systems*. Oslo, Norway: Kluwer. 149–161.

Coenen, F., and Bench-Capon, T. 1993. *Maintenance of Knowledge-Based Systems*. Academic Press.

Fowler, M. 1999. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley.

Groot, P.; van Harmelen, F.; and ten Teije, A. 2000. Torture Tests: A Quantitative Analysis for the Robustness of Knowledge-Based Systems. In *Knowledge Acquisition, Modeling and Management, LNAI 1319*, 403–418. Berlin: Springer Verlag.

Knauf, R.; Philippow, I.; Gonzalez, A. J.; Jantke, K. P.; and Salecker, D. 2002. System Refinement in Practice – Using a Formal Method to Modify Real-Life Knowledge. In *Proceedings of 15th International Conference - Florida Artificial Intelligence Research Society (FLAIRS-2002)*, 216–220.

Knauf, R. 2000. *Validating Rule-Based Systems: A Complete Methodology*. Aachen, Germany: Shaker.

Lorenz, K.-W.; Baumeister, J.; Greim, C.; Roewer, N.; and Puppe, F. 2003. QualiTEE - An Intelligent Guidance and Diagnosis System for the Documentation of Transesophageal Echocardiography Examinations. In *Proceedings of the 14th Annual Meeting of the European Society for Computing and Technology in Anaesthesia and Intensive Care (ESCTAIC)*.

Menzies, T. 1999. Knowledge Maintenance: The State of the Art. *The Knowledge Engineering Review* 14(1):1–46.

Opdyke, W. F. 1992. *Refactoring Object-Oriented Frameworks*. Ph.D. Dissertation, University of Illinois, Urbana-Champaign, IL, USA.

Preece, A.; Shinghal, R.; and Batarekh, A. 1992. Verifying Expert Systems. A Logical Framework and a Practical Tool. *Expert Systems with Applications* 5(3/4):421–436.

Preece, A. 1998. Building the Right System Right. In *Proceedings of KAW'98 Eleventh Workshop on Knowledge Acquisition, Modeling and Management*.

Schreiber, G.; Akkermans, H.; Anjewierden, A.; de Hoog, R.; Shadbolt, N.; de Velde, W. V.; and Wielinga, B. 2001. *Knowledge Engineering and Management - The CommonKADS Methodology*. MIT Press, 2 edition.