

# Prioritized Use Cases as a Vehicle for Software Inspections

**Thomas Thelin and Per Runeson**, *Lund University*

**Claes Wohlin**, *Blekinge Institute of Technology*

**S**oftware inspection has garnered much attention since Michael Fa- gan introduced it in 1976.<sup>1</sup> Inspection, a static verification and validation technique, is applicable to all kinds of software development artifacts, because inspections don't require dynamic execution. It is a common industry practice in domains where the delivered software product quality is important. Although researchers have studied inspection a lot, the traditional checklist-based reading (CBR) approach still dominates industry.<sup>2,3</sup>

Another software engineering trend over the last two decades has been object-oriented design methods and the application of use cases in this context.<sup>4</sup> *Scenarios* (sequences of steps describing an interaction between a user and a program) and *use cases* (sets of scenarios tied together by a common user goal) help developers focus on users' needs instead of on technology alone.<sup>5</sup> For example, developers can document use cases and scenarios with UML use case diagrams and sequence diagrams or by using struc-

ture text. Use cases are typically used in specifying a system but can also be used for inspection purposes later in the development process.

Usage-based testing, or operational-profile testing, helps focus test activities on a system's users and usage.<sup>6</sup> The testers select the test cases according to the frequency or probability of operational use. So, testers check the most important user functions first and devote the most time to them. The typical inspection process<sup>7</sup> involves

1. A group of reviewers overviewing the artifact to be inspected
2. Each reviewer reading the artifact (individual preparation)
3. Compiling the findings into one list of issues
4. Conducting an inspection meeting
5. Reworking and following up

**The usage-based reading technique combines traditional inspection principles, use cases, and operational profile testing to create an efficient, user-oriented software inspection reading technique. UBR can find faults more effectively and efficiently than the traditional checklist-based method.**

We combine inspections, use cases, scenarios, and principles of operational-profile testing into *usage-based reading* to provide an efficient reading technique for software design inspections.<sup>8,9</sup> UBR considers that different faults can affect the user's perceptions of a product differently. The technique focuses primarily on design inspections, but it is useful for requirements and code inspections as well as for test case development; furthermore, other sources have discussed use-case-guided code inspections.<sup>10</sup>

### Usage-based reading

UBR is useful for documents developed after use case derivation, such as requirements, design, code, and test documents. It assumes that the use cases and scenarios have been defined earlier in the development process. UBR utilizes the set of use cases to focus the inspection effort, just as test cases focus the test effort.

The method's basic steps are as follows (see Figure 1):

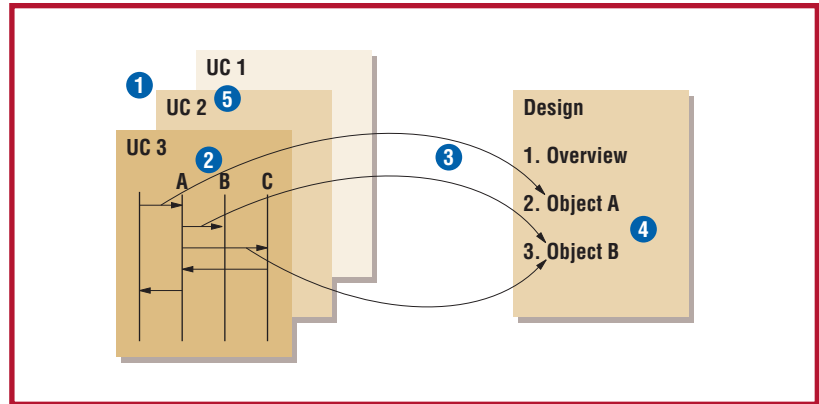
Preliminarily,

1. Prioritize the use cases in order of importance from a user perspective.

In the preparation phase of inspections,

2. Select the use case with the highest priority.
3. Track the use case's scenarios through the document under inspection.
4. During tracking, ensure that the document under inspection fulfills the use case's goal; for example, make sure it provides the needed functionality and that the interfaces are correct. Identify and report the issues that tracking reveals.
5. Select the next use case and repeat from step 3 until the time is up or you've covered all use cases.

Because the UBR method's steps cover only the preparation part of the inspection process, it works as an add-on to existing inspection processes. It fits well with established resource-scheduling, meeting, and follow-up procedures. Developers can introduce the method easily in a project in which the use cases and scenarios have sufficient detail and completeness. Although detailed scenarios are better than use case descriptions alone, it is not beneficial to develop the scenarios solely for the UBR inspection.<sup>11</sup>



**Figure 1. Usage-based reading.** The letters in the use case document correspond to those in the design document. So, when something is happening with object A in the use case, the reviewer checks it in the corresponding section in the design document. For example, if A is a taxi and B a central unit, there will be communication signals between A and B.

If the system has a defined operational profile, it contains important information for the prioritization activity. Otherwise, reviewers can use pairwise comparisons guided by the *analytic hierarchy process* to perform the prioritization.<sup>12</sup> The AHP helps users determine the consistency of the comparisons (if all pairs have been compared). Users can also use it to find priorities without having to explicitly define the relations between all pairs of use cases.

### Experimental evaluation

We evaluated UBR in three experiments.<sup>13</sup> The second one, described here, compared UBR to the industry-standard CBR, and revealed that UBR is preferable.<sup>9</sup>

#### The experimental setting

We launched an experiment to compare UBR's and CBR's *effectiveness* (share of faults found) and *efficiency* (faults found per time unit) for individual preparation.<sup>9</sup> Eleven students in the last year of a software engineering master's program applied UBR and 12 applied CBR to inspect a design document for a taxi management system.

The system manages a fleet of taxis, dispatching customer orders and following up on each driver's transports. Working with actual stakeholders, we created an academic product by scaling down the system from its real application domain. The design document contains 2,300 words and 38 faults ranked A (*crucial*),

**Table 1****Efficiency data (faults found per hour)**

Fault class	Mean		Standard deviation	
	UBR	CBR	UBR	CBR
All faults*	5.6	4.1	2.0	2.0
Class A faults*	2.6	1.3	1.0	1.1
Class B faults	2.1	1.4	1.2	0.7
Class C faults	0.9	1.4	0.4	0.8
Class A + Class B faults*	4.7	2.8	1.8	1.7

\*Statistically significant at a 95% level

**Table 2****Effectiveness data (share of faults found)**

Fault class	Mean		Standard deviation	
	UBR	CBR	UBR	CBR
All faults (38)	0.31	0.25	0.09	0.14
Class A faults (13)*	0.43	0.24	0.17	0.21
Class B faults (14)	0.31	0.24	0.15	0.13
Class C faults (11)	0.18	0.30	0.08	0.21
Class A + Class B faults (27)*	0.37	0.24	0.12	0.16

\*Statistically significant at a 95% level

**Table 3****Preparation and inspection time (minutes)**

Fault class	Mean		Standard deviation	
	UBR	CBR	UBR	CBR
Preparation	53	59	20	15
Inspection	77	81	18	19
Total	130	140	15	12

B (*important*), or C (*unimportant*) from a user's perspective. Thirteen faults were ranked A, 14 B, and 11 C. Before the experiment, we reinserted 28 faults that had appeared during the design document's initial development. The system's developer seeded eight new faults, and during the experiment, the students found two additional faults from the original development, for 38 faults total. The use case document comprised 24 use cases with defined scenario sequences and alternatives. Different experts prioritized the use cases and the faults to ensure independent priorities.

The students learned the reading technique they were going to use before we launched the experiment. They had enough knowledge and experience to represent industrial programmers

accurately. Many of the students had industrial software engineering experience and all of them had participated in a full-semester, 15-member project with real customers. We administered a pretest survey to ensure that the two groups had comparable experience and skill.

**Experimental results**

The experiment shows that UBR is significantly more efficient and effective than CBR. It finds more faults per time unit for crucial and important faults (classes A and B) and finds a larger share of the faults. Tables 1 and 2 present the data, and Table 3 presents preparation and inspection time. The differences denoted with an asterisk are statistically significant on a 95 percent level. More details on the data and its analysis are available elsewhere.<sup>9</sup>

The UBR reviewers spent on average 6.5 minutes less in preparation and 4 minutes less in inspection. Nevertheless, they found more faults.

Regarding efficiency, UBR reviewers found twice as many crucial (class A) faults per hour (2.6 versus 1.3 A faults) as CBR reviewers did, and about 50 percent more of the important (class B) faults per hour (2.1 versus 1.4 class B faults).

Regarding effectiveness, reviewers using UBR identified on average 21 percent more faults (0.31 versus 0.25) and 75 percent more crucial faults (0.43 versus 0.24 class A faults) than reviewers using CBR did. CBR, however, discovered 63 percent more of the unimportant faults (0.30 versus 0.18 class C faults). (Efficiency and effectiveness percentages may not total 100 percent due to rounding.)

The data reveals that the UBR method succeeds in directing the inspection effort toward the most important problems from a user's perspective. CBR, conversely, wastes effort searching for issues.

**U**BR's main purpose is to focus inspections on the users' needs. Adding a time limit to the prioritized use case provides control over the reviewers' time consumption, allowing reviewers to predict the inspection duration. Even with limited time, reviewers detect the most critical faults.

Usage-based testing is used with the same purpose as UBR, but in the testing phase.<sup>6</sup> Hence, UBR and UBT are complementary

fault-detection techniques in software development. Both focus on the product's end users. A software organization could employ both techniques in the development process to place the principal focus on the users from the start.

The UBR method is based on information that is available in many development projects—that is, use cases. The additional effort needed is the time to prioritize the use cases to guide the inspection effort. Because the UBR method integrates easily into existing inspection processes, we hope it will provide support to software professionals striving toward better quality and more-efficient engineering methods. ☉

## References

1. M.E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM System J*, vol. 15, no. 3, 1976, pp. 182–211.
2. A. Aurum, H. Petersson, and C. Wohlin, "State-of-the-Art: Software Inspections after 25 Years," *Software Testing Verification and Reliability*, vol. 12, no. 3, Sept. 2002, pp. 133–154.
3. O. Laitenberger and J.-M. DeBaud, "An Encompassing Life Cycle Centric Survey of Software Inspection," *J. Systems and Software*, vol. 50, no. 1, Jan. 2000, pp. 5–31.
4. I. Jacobson et al., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.
5. M. Fowler and K. Scott, *UML Distilled*, Addison-Wesley, 2000.
6. J.D. Musa, "Operational Profiles in Software-Reliability Engineering," *IEEE Software*, vol. 10, no. 2, Mar. 1993, pp. 14–32.
7. A.F. Ackerman, L.S. Buchwald, and F.H. Lewski, "Software Inspections: An Effective Verification Process," *IEEE Software*, vol. 6, no. 3, May 1989, pp. 31–36.
8. T. Thelin, P. Runeson, and B. Regnell, "Usage-Based Reading—An Experiment to Guide Reviewers with Use Cases," *Information and Software Technology*, vol. 43, no. 15, Dec. 2001, pp. 925–938.
9. T. Thelin, P. Runeson, and C. Wohlin, "An Experimental Comparison of Usage-Based and Checklist-Based Reading," to be published in *IEEE Trans. Software Eng.*, vol. 29, no. 8, Aug. 2003.
10. A. Dunsmore, M. Roper, and M. Wood, "Further Investigation into the Development and Evaluation of Reading Techniques for Object-Oriented Code Inspection," *Proc. Int'l Conf. Software Eng. (ICSE 02)*, ACM Press, 2002, pp. 47–57.
11. T. Thelin et al., "How Much Information Is Needed for Usage-Based Reading?—A Series of Experiments," *Proc. Int'l Symp. Empirical Software Eng. (ISESE 02)*, IEEE CS Press, 2002, pp. 127–138.
12. T.L. Saaty and L.G. Vargas, *Models, Methods, Concepts & Applications of the Analytic Hierarchy Process*, Kluwer Academic Publishers, 2001.
13. C. Wohlin et al., *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, 2000.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.

## About the Authors



**Thomas Thelin** is an associate professor of software engineering in the Department of Communication Systems at Lund University. His research interests include empirical methods in software engineering; software quality; and verification and validation with emphasis on testing, inspections, and estimation methods. He is working on the European project MaTeLo, which seeks to develop an automatic generator of test cases derived from Markov usage models. He has a PhD in software engineering from Lund University and is a member of the IEEE. Contact him at Dept. of Communication Systems, Lund Univ., Box 118, SE-22100 Lund, Sweden; [thomas.thelin@telecom.lth.se](mailto:thomas.thelin@telecom.lth.se); <http://serg.telecom.lth.se/personnel/thomast>.

**Per Runeson** is an associate professor of software engineering in the Department of Communication Systems, Lund University, and has been the leader of the Software Engineering Research Group since 2001. His research interests include software development methods and processes, in particular methods for verification and validation, with special focus on efficient and effective methods to facilitate software quality. He received a PhD in communication systems from Lund University. He is a senior member of the IEEE and a member of the ACM. Contact him at Dept. of Communication Systems, Lund Univ., Box 118, SE-22100 Lund, Sweden; [per.runeson@telecom.lth.se](mailto:per.runeson@telecom.lth.se).



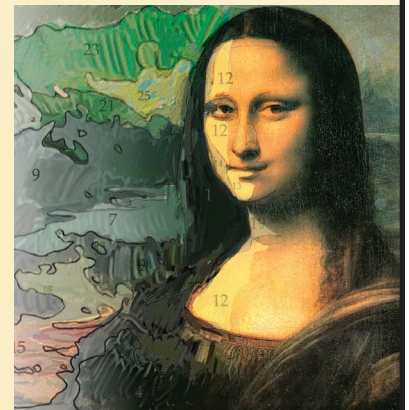
**Claes Wohlin** is a professor of software engineering in the Department of Software Engineering and Computer Science at Blekinge Institute of Technology. His research interests include empirical methods in software engineering, software metrics, software quality, and systematic improvement in software engineering. He is coeditor in chief of *Information and Software Technology* and is on the editorial boards of *Empirical Software Engineering: An International Journal* and the *Software Quality Journal*. He has a PhD in communication systems from Lund University. He is a member of the IEEE and ACM. Contact him at Blekinge Institute of Technology, Box 520, SE-372 25 Ronneby, Sweden; [claus.wohlin@bth.se](mailto:claus.wohlin@bth.se).

# Master software with these future topics:

Developing with Open Source Software

Model-Driven Development

The State of the Practice of Software Engineering



IEEE  
**Software**

Visit us on the web at

<http://computer.org/software>