

A Lightweight Sanity Check for Implemented Architectures

Eric Bouwers, *Software Improvement Group*

Arie van Deursen, *Delft University of Technology*

The lightweight sanity check for implemented architectures (LiSCIA) is an easy-to-use evaluation method that can reveal potential problems as a software system evolves.

Software architecture has been loosely defined as the organizational structure of a software system, including the components, connectors, constraints, and rationale.¹ Evaluating a system's software architecture helps stakeholders to check whether the architecture complies with their interests. Additionally, the evaluation can result in a common understanding of the architecture's strengths and weaknesses. All of this helps to determine which quality criteria the system meets because "architectures allow or preclude nearly all of the system's quality attributes."²

Many architecture evaluation methods are available (see the sidebar, "Related Work on Architecture Evaluation and Erosion").^{3,4} Unfortunately, as a survey conducted by Muhammad Babar and Ian Gorton showed, industry's adoption of architecture evaluations is low. Babar and Gorton concluded that "there is limited out-of-the-box process and tool support for companies that want to start doing architecture evaluations."⁵

We propose bridging this gap through a *lightweight sanity check for implemented architectures* (LiSCIA), based on nine years of experience in evaluating more than 100 different industrial software systems, as well as on our earlier research on maintainability indicators.⁶⁻⁸ LiSCIA is a concrete, easy-to-apply architecture evaluation method that aims to obtain insight into a system's quality within a day. By applying LiSCIA at the start of a software project and then periodically (for example, every six months or at every release),

evaluators can spot potential problems with the implemented architecture quickly and deal with them at an early stage.

Evaluating Architectures

Existing methodologies for architecture evaluations are divided into early and so-called late evaluations.⁴ Early evaluations focus on designed architectures, whereas late architecture evaluations focus on an architecture after it has been implemented. LiSCIA falls in the latter category as it aims to evaluate an implemented architecture.

Our experience shows that recurrent evaluation of an implemented architecture helps to identify *architecture erosion*,⁹ the steady decay of an implemented architecture's quality. In past years, the Software Improvement Group (SIG; www.sig.eu) has offered this type of recurrent evaluation as part of its software monitoring service¹⁰ and software risk assessments (SRAs).⁷

In both services, the SIG examines a system's technical quality and links it to business risks. An SRA does this once, whereas a software monitor follows a system's evolution over a longer period of time.

Recently, we conducted a study using more than 40 risk assessment reports from the past two years. We identified 15 system attributes that influence the quality of an implemented architecture.⁶ These 15 attributes, together with our experience in monitoring the development of software systems, form the basis of LiSCIA. In essence, we designed LiSCIA to concretely measure these abstract attributes. Additionally, LiSCIA represents a basic formalization of the steps we normally undertake at the start of an SRA and during the re-evaluations within a software monitoring project.

LiSCIA focuses on the "maintainability" quality attribute of a software system. Because of its lightweight nature, it doesn't offer a complete architecture evaluation. However, by recurrently applying LiSCIA, evaluators can obtain insights into the current status of a system's implemented architecture. With this recurring insight, evaluators can take appropriate actions to control the architecture's erosion. Additionally, LiSCIA's results offer a platform for discussing current issues and can justify refactorings or a broader architecture evaluation.

LiSCIA Design

While designing LiSCIA, we considered the following key issues to ensure that it's practical yet generally applicable:

- The evaluation takes no more than a day.
- The evaluation includes ways to improve the system—that is, it helps the evaluator define actions.
- The evaluation isn't limited to a specific programming language or technology.
- The evaluation can handle different levels of abstraction.

LiSCIA consists of two phases: a *start-up* phase (done once) and an *evaluation* phase (performed for every evaluation). The start-up phase results in an overview report, which is the input for the evaluation phase. The evaluation phase produces an evaluation report containing the results of the evaluation and specific actions. These actions might require adjustments to the overview report. Both the (possibly adjusted) overview report and the evaluation report serve as input to a system reevaluation. Figure 1 illustrates the complete process.

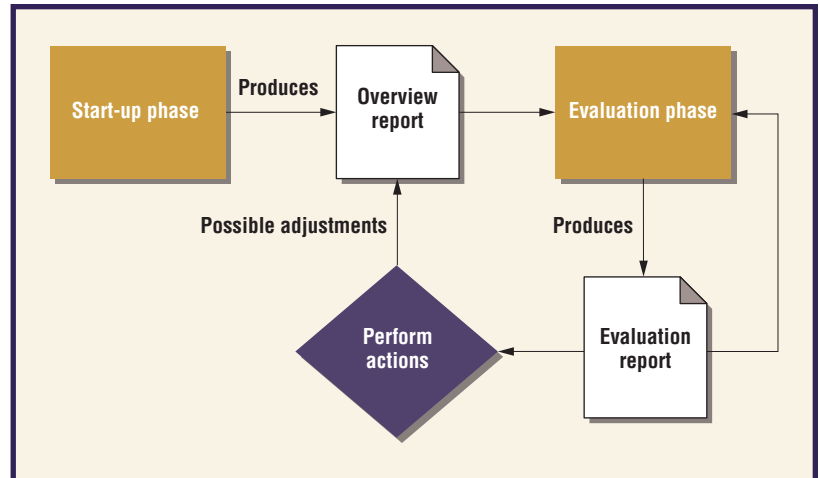


Figure 1. Overall flow of the lightweight sanity check for implemented architectures (LiSCIA) evaluation method. After the definition of an overview report, the evaluation of the implemented architecture results in an evaluation report. This report serves both as a guideline for actions to improve the implemented architecture and as a start for future evaluations.

Definitions

LiSCIA uses the *module* viewtype to reason about an implemented architecture's structure.¹¹ This viewtype divides the system into coherent chunks of functionality called modules. A module can represent some business functionality, such as accounting or stocks, or a more technical functionality, such as GUI or XML processing. We can apply LiSCIA to both decompositions. Better yet, we can apply LiSCIA to the same version of a system using different decompositions. In this way, we can explore different modularization views on the architecture. Each view can give different insights, which can lead to a better understanding of the implemented architecture as a whole.

A *unit* within LiSCIA is a logical block of source code that implements some sort of functionality. The typical unit in LiSCIA is a source file. Units are grouped into *containers*. In LiSCIA, the normal container is a directory on the file system.

Using the source file as a unit complies with the notion that files are typically the dominant decomposition of functionality.¹² In other words, most programming languages use the file as a logical grouping of functionality. An argument against this decomposition is that some technologies offer a more fine-grained granularity of functionality. For example, for the Java language, the classes (or even methods) can be seen as a separate decomposition of functionality. We chose files to make the method independent of the evaluated technology.

Table 1**Key properties of the categories of the lightweight sanity check for implemented architectures (LiSCIA)**

Name	Topics of interest	No. of questions	No. of actions
Source groups	Current grouping of units in modules and future modules	4	4
Module functionality	Decomposition of functionality over modules	5	6
Module size	Size of modules, distribution of system size over modules, and growth of modules	6	5
Module dependencies	Expected, circular, unwanted, and changed dependencies	6	7
Technologies	Combination, version, usage, and size distribution of the used technologies	7	6

Start-up Phase

In the start-up phase, the evaluator must first define the modules of the system under review. In some cases, the modules are defined in the technical documentation; in others, the modules are apparent from, for example, the directory, package, or namespace structure. When high-level documentation is unavailable, the evaluator can obtain modules by interviewing the system's developers. In our experience, developers can typically produce a description (and drawing) of the modules (and the relationships between them) of the systems they work on. Details in these descriptions are often incorrect, but adjustments to the descriptions can be defined as actions in the evaluation phase.

After defining the modules, the evaluator should place each unit in the system under one of the defined modules. To do this, patterns are placed on the names of the units and their containers. For example, if all units in the "gui" container are part of the GUI module, a logical pattern for the module would be `*/gui/*`.

To ensure a well-balanced evaluation, every unit should be matched to a single module. A unit that should actually be placed under multiple modules likely implements parts of different functionalities, which indicates limited separation of concerns. In this case, the developers should either split these units into different parts, or the evaluator should introduce a module capturing the two functionalities.

As a last step in the start-up phase, the evaluator should identify the types of technologies used within the project. For LiSCIA, "technologies" refers not only to programming languages, but also to frameworks, libraries, build tools, and possibly even hardware platforms.

The start-up phase report includes a description of the modules and patterns and a list of technologies.

Evaluation Phase

The LiSCIA evaluation phase consists of answering a list of questions concerning the architectural elements identified in the start-up phase. Many of the questions are grouped into pairs. Usually, the first question asks about a specific situation and the second requests an explanation. The answer to the first question is either "yes" or "no," whereas the answer to the second question is open ended. This set-up requires the evaluator to be explicit, but leaves room for explaining why a certain situation occurs.

LiSCIA also provides a set of actions linked to the questions that can serve as a guide to answering the questions. In principle, the answers must explain why the action belonging to a question doesn't need to be taken. We defined the actions such that when there's no valid reason to ignore the action, the implemented architecture's maintainability can benefit from performing the given action.

We divide the questions and actions into five categories. These categories cover the grouping of sources, the technologies used in the system, and the modules' functionalities, sizes, and dependencies. Table 1 lists some key properties of these categories. The complete list of 28 questions and 28 actions appears at www.sig.eu/en/liscia.

Each category includes questions related to the current situation as well as questions related to the previous evaluation. The latter type of questions can be ignored in the initial evaluation, since their primary objective is to reveal the reasons for differences between the versions compared.

Source groups. The "source groups" category has a good example of reevaluation questions. As a first step in this category, the evaluator must determine whether all units belong to a module, given the patterns described in the overview report. During the

start-up phase, the evaluator designs the patterns for the module to capture all units. It's therefore likely that during the first evaluation, all units are placed under a module. Because most of the questions in this category are about units that aren't matched to a module, they can be ignored for this first evaluation session.

During later evaluation sessions, an evaluator might find that new units have been added to the system that aren't matched by any module pattern. We've experienced this situation many times. In some cases, these units are simply mistakenly misplaced and the units can easily be moved to their correct locations. In other cases, a new module (together with a new pattern) must be introduced because new functionality is introduced. Questions regarding the ideas behind this new module, and possibly additional modules, are also part of this category of questions.

Even though the mapping of units to modules should be complete during the first evaluation, this isn't always the case, especially when the evaluator uses existing documentation as part of the overview report. For example, in one of our assessments, we evaluated a system containing more than 4 million lines of code. The existing documentation described several modules. In addition, the documentation included a file describing each source file's mapping to one of these modules. Evaluating this mapping carefully, we discovered that more than 30 percent of the source files couldn't be assigned to a module. Additionally, a considerable part of the mapping contained source files that no longer existed. One of the actions defined for this system was to reorder the source files to better resemble the module structure as outlined in the documentation. This example illustrates the importance of verifying existing documentation against the current implementation, instead of assuming that the documentation is correct.

Module functionality. In the second category, the evaluator focuses on how the system's functionality is spread out over the modules. For example, one question asks whether each module's functionality can be described in a single sentence. The question helps the evaluator determine whether the current decomposition of functionality isn't too generic. Several systems we encountered defined a module called "utilities" (or something similar). When this module's exact functionality is described, it turns out that the module contains not only generic functionality, but also business functionality, specific parsing functionality, or an object model. In these cases, the action is to split up the module into

Related Work on Architecture Evaluation and Erosion

A major distinction exists between the lightweight sanity check for implemented architectures (LiSCIA) and other architecture evaluation techniques.^{1,2} Unlike many of the available architecture evaluations, LiSCIA predefines a notion of quality in terms of maintainability. Other available architecture evaluation methods, including those explicitly aimed at an implemented architecture, such as the one proposed by Per Olof Bengtsson and Jan Bosch,³ don't provide such a notion. Instead, virtually all methods contain a phase in which the evaluators define the notion of quality. As we discuss in the main text, this limits the use of LiSCIA to a specific purpose, but makes it easier to start an architecture evaluation.

Approaches for dealing with architecture erosion typically try to incorporate a solution into the architecture's design. This approach helps avoid architecture erosion, but Jilles van Gorp and Jan Bosch conclude that "even an optimal design strategy for the design phase does not deliver an optimal design."⁴ In addition, predicting all new and changed requirements during the definition of a first release of a system is impossible. So, even if the design was optimal in some sense for the first release, there's a good chance that the design needs to be adapted.

To conclude, you can't completely avoid changes to an implemented architecture. Therefore, an evaluation should account for the architecture that is currently implemented when dealing with software change to avoid, or minimize, architectural erosion.

Some approaches do account for the implemented architecture, such as the approach Nenad Medvidovic and Vladimir Jakobac proposed.⁵ The main difference between this (and similar) approaches and LiSCIA is when they deal with erosion. LiSCIA tries to detect erosion when it has happened, whereas other approaches try to prevent erosion from happening. Because these approaches are complementary, both are useful and necessary. We realize that dealing with erosion after it has happened is more difficult and costly, but it's better to deal with erosion as soon as it's been introduced rather than when other issues need to be resolved.

References

1. M.A. Babar, L. Zhu, and D.R. Jeffery, "A Framework for Classifying and Comparing Software Architecture Evaluation Methods," *Proc. 2004 Australian Software Eng. Conf. (ASWEC 04)*, IEEE CS Press, 2004, pp. 309–319.
2. L. Dobrica and E. Niemelä, "A Survey on Software Architecture Analysis Methods," *IEEE Trans. Software Eng.*, vol. 28, no. 7, 2002, pp. 638–653.
3. P.O. Bengtsson and J. Bosch, "Scenario-Based Software Architecture Reengineering," *Proc. 5th Int'l Conf. Software Reuse (ICSR 98)*, IEEE CS Press, 1998, pp. 308–317.
4. J. van Gorp and J. Bosch, "Design Erosion: Problems and Causes," *J. Systems and Software*, vol. 61, no. 2, 2002, pp. 105–119.
5. N. Medvidovic and V. Jakobac, "Using Software Evolution to Focus Architectural Recovery," *Automated Software Eng.*, vol. 13, no. 2, 2006, pp. 225–256.

a truly generic part and separate modules for the more specific types of functionality.

Module size. This third category of questions relates to module size. To keep LiSCIA usable for a large range of systems, we intentionally kept the exact definition of "size of a module" abstract. Nevertheless, in most cases the size of a module

**Developers
or project
managers
can use the
evaluation
phase report
to justify
certain
refactorings.**

can be represented by the sum of the LOC of all units in the module.

The questions in this category aren't related only to the individual modules' size, they also consider the distribution of the size of the complete system over the modules. Additionally, this category contains questions related to the modules' growth. These last types of questions are especially useful in detecting unwanted evolution within the system, but also help in detecting unwanted development effort.

For example, in one of our monitoring projects, several modules were marked as "old." These old modules contained poor-quality legacy code that was still used, but wasn't actively maintained. After inspecting the modules' growth, we discovered that new functionality was still being added to these old modules. The explanation given was that it was easier to add the new functionality in this module. However, even though it was easier, it resulted in the addition of large and complex pieces of code because this new code had to follow the legacy code's structure. The action that resulted from this observation was the functionality's migration to a new module, where it would be easier to maintain and test.

Module dependencies. To answer the questions in this category, the dependencies between modules must be available. Similar to module size, we kept the "dependencies between modules" concept abstract. However, we can calculate the dependencies between modules by first determining the dependencies between units (for example, the calls between methods inside the source files), after which we can raise these dependencies to the module level.

The questions in this category help the evaluator assess the wanted, unwanted, and circular dependencies between modules. In addition, we defined questions about added and removed dependencies for when a previous evaluation is available. One of the questions in the latter category is whether any new dependencies have been added, as was the case on one of our monitoring projects. The follow-up question is whether this dependency is expected, which, in the case of this project, it wasn't. The new dependency wasn't allowed according to predefined layering rules. After reporting this violation of the architecture, the project lead was surprised and asked the developers for an explanation. The developers recognized the violation and explained that it was introduced because a third module wasn't finished yet. As soon as this third module was finished, the dependency

would be removed. An evaluation report documented all of this. Four months later, when the third module was finished, the evaluator reminded the developers of this undesired dependency and they removed it.

Technologies. The last category helps the evaluator assess the combination of the technologies used within the system. In addition, the questions in this category deal with the age, usage, and support for each of the technologies. For example, one question asks whether the latest version of each technology is used. In many projects, the latest version isn't used. The usual explanation for this is that developers had no time to upgrade the system. In these situations, the evaluator defines an action to upgrade to the latest technology as soon as possible. This is done to prevent developers from using legacy technology without an easy upgrade path. A different explanation for the same situation is that management decided to always use the second-to-last version of a technology because this version has already proven itself in practice. In these cases, no action is defined, but the evaluation report documents the explanation for not implementing the action.

Result. The evaluation phase report documents answers to all the questions and, if applicable, a list of actions. With this report, developers or project managers can justify certain refactorings. In addition, the report gives a basic overview of the architecture as it's currently implemented. The report can therefore serve as a factual basis for discussions about the current architecture and as input to the next evaluation session.

Applications

Because LiSCIA is based on our experience in monitoring existing software systems, we were able to provide examples to show how evaluators can use the different parts of the method to discover problems in an implemented architecture. To evaluate the complete LiSCIA method, we're currently using LiSCIA as part of our daily practice. Currently, we don't have enough data available to formally evaluate the method, but the first results look promising.

We've applied LiSCIA in several situations, such as to start the evaluation process of a new system and as a sanity check after an evaluation. Overall, the consultants who used LiSCIA reacted positively. They appreciated the methodology's structure, as well as the questions' type and ordering. One consultant was particularly pleased be-

cause LiSCIA's straightforward application has, in his words, "mercilessly led to the discovery of the embedding of a forked open source system in the code."

Apart from this discovery, LiSCIA's application led to more detailed evaluations of certain system attributes. For example, the addition of a new technology in one system led to a detailed evaluation of how this new technology interacted with the old technologies, and how the old technologies interacted with each other. In a different system, LiSCIA's application warranted a deeper look into how functionality was divided over the modules. In general, LiSCIA was able to identify which attributes of the implemented architecture deserved a more thorough evaluation.

The application of LiSCIA took between 30 minutes (for the sanity check) and one day (as the start of a new system's evaluation). In the first case, all data needed to evaluate the system was already available, while in the latter case, the data was constructed during LiSCIA's application. In both cases, the consultants weren't involved in the reviewed systems' development.

These preliminary results show that we can apply LiSCIA within a day and that it helps identify weaknesses in the implemented architecture. However, we must conduct a more formal study to confirm these results.

Limitations

The initial evaluation of LiSCIA brought some critical aspects of the method to our attention. As a start, one consultant stressed that LiSCIA should be applied by an expert outside the development team, pointing out that "it is hard for a software engineer to remain objective when it concerns his own code."

LiSCIA does indeed rely heavily on the evaluator's opinion. After all, it is the evaluator who decides whether a given explanation is good enough in the project's setting. Therefore, we believe that it's a good idea to let a second expert examine the evaluation report, just to make sure that none of the explanations are flawed.

Different consultants also pointed out a second limitation of LiSCIA. They stated that the method relies on some of our internal toolset's functionality, which might limit the usability outside of our company. This is a correct observation, since LiSCIA relies on (automated) tool support to determine, for example, the size of the modules and the dependencies between the system's units. However, we believe that anyone can make these types of measurements through freely available tools.

About the Authors



Eric Bouwers is a software engineer at the Software Improvement Group and a part-time PhD student in computer science at Delft University of Technology. His research interests include the architectural and linguistic aspects of software quality. Bouwers has a master of science in computer science from Utrecht University. Contact him at e.bouwers@sig.eu.

Arie van Deursen is a full professor of software engineering at Delft University of Technology, where he leads the Software Engineering Research Group. His research interests include software testing, software architecture, program comprehension, and the use of Web 2.0 techniques in software engineering. van Deursen has a PhD in computer science from the University of Amsterdam. Contact him at Arie.vanDeursen@tudelft.nl.




The only investment needed is in the aggregation of the raw output of these tools, which is a relatively minor investment.

A third limitation of LiSCIA is that it only aims to discover potential risks related to maintainability. Additionally, because LiSCIA uses only a single viewpoint to evaluate the architecture, it likely doesn't even cover all potential risks in this area.

In our view, these limitations can also be considered the method's strengths. First of all, when a system isn't maintainable, dealing with other quality issues such as performance and reliability becomes more difficult. Thus, a first focus on maintainability is justified. This focus also allows for a scoped setup of the questionnaire, which makes sure that implementing LiSCIA in current projects only requires little effort. Moreover, we deliberately positioned LiSCIA as a lightweight check to ensure that it's seen as a stepping stone toward broader architecture evaluations. Last but not least, the collection of questions and actions reflects years of experience in conducting architectural evaluations, making it likely that they cover the most important maintainability risks.

We're currently evaluating the formal LiSCIA method by applying it in our current practice. In addition, we're interested to see whether LiSCIA is useful in environments outside SIG. For this, we call upon you to try out LiSCIA and share your results with us. By combining our own experience with community feedback, we hope to report on

an improved version of LiSCIA in the coming year. The complete LiSCIA method is available at www.sig.eu/en/liscia. 

Acknowledgments

We thank all of our colleagues at the Software Improvement Group for their time and contributions to the making of the LiSCIA methodology and this article.

References

1. P. Clements and P. Kogut, "The Software Architecture Renaissance," *Crosstalk—The J. Defense Software Eng.*, vol. 7, 1994, pp. 20–24.
2. P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures*, Addison-Wesley, 2005.
3. M.A. Babar, L. Zhu, and D.R. Jeffery, "A Framework for Classifying and Comparing Software Architecture Evaluation Methods," *Proc. 2004 Australian Software Eng. Conf. (ASWEC 04)*, IEEE CS Press, 2004, pp. 309–319.
4. L. Dobrica and E. Niemelä, "A Survey on Software Architecture Analysis Methods," *IEEE Trans. Software Eng.*, vol. 28, no. 7, 2002, pp. 638–653.
5. M.A. Babar and I. Gorton, "Software Architecture Review: The State of Practice," *Computer*, vol. 42, no. 7, 2009, pp. 26–32.
6. E. Bouwers, J. Visser, and A. van Deursen, "Criteria for the Evaluation of Implemented Architectures," *Proc. 25th Int'l Conf. Software Maintenance (ICSM 2009)*, IEEE CS Press, 2009, pp. 73–83.
7. A. van Deursen and T. Kuipers, "Source-Based Software Risk Assessment," *Proc. Int'l Conf. Software Maintenance (ICSM 03)*, IEEE CS Press, 2003, pp. 385–388.
8. I. Heitlager, T. Kuipers, and J. Visser, "A Practical Model for Measuring Maintainability," *Proc. 6th Int'l Conf. Quality of Information and Comm. Technology (Quatic 07)*, IEEE CS Press, 2007, pp. 30–39.
9. D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *SIGSOFT Software Eng. Notes*, vol. 17, no. 4, 1992, pp. 40–52.
10. T. Kuipers and J. Visser, "A Tool-Based Methodology for Software Portfolio Monitoring," *Software Audit and Metrics*, Inst. for Systems and Technologies of Information, Control, and Comm. (INSTICC) Press, 2004, pp. 118–128.
11. P. Clements et al., *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, 2003.
12. P. Tarr et al., "N Degrees of Separation: Multi-dimensional Separation of Concerns," *Proc. 21st Int'l Conf. Software Eng. (ICSE 99)*, ACM Press, 1999, pp. 107–119.

CALL FOR ARTICLES

Software Components beyond Programming—From Routines to Services

FINAL SUBMISSIONS DUE: 1 NOVEMBER 2010

PUBLICATION DATE: MAY/JUNE 2011

Components have been known in software engineering since 1968 and have had several revivals, including component-based software engineering and service-oriented architectures. These approaches brought new expectations and advanced the current state of art and practice, but also left unfulfilled promises.

This special issue aims to provide an overview of what component software has contributed to software engineering, what challenges have been addressed, which have been solved and which not, and what is the state of the art and current industrial practice. We plan to show successes, experiences, and possibly failures in component-based approaches in practice. Our aim is also to bring about a discussion of different aspects of component software used in different application domains—such as information systems, or embedded systems—and also in different software engineering domains—such as SOA, product line architectures, and so on. We especially welcome case studies, lessons learned, and success and failure stories in component software. The issue will summarize current results and expectations for the near future.

POSSIBLE TOPICS INCLUDE BUT ARE NOT LIMITED TO

- CBSE's achievements
- The main concerns and challenges of component software

- Components in different areas of SE
- Component software in different business and application domains
- Transformation of legacy systems to components and services—state of the practice
- Practical solutions on the horizon
- Future research directions

FOR MORE INFORMATION ABOUT THE FOCUS, CONTACT THE GUEST EDITORS:

- Ivica Crnkovic, Mälardalen University, Sweden, ivica.crnkovic@mdh.se
- Judith Stafford, Tufts University, US, jas@cs.tufts.edu
- Clemens Szyperski, Microsoft, clemens.szyperski@microsoft.com

For complete CFP: www.computer.org/software/cfp3

For author guidelines: www.computer.org/software/author

For submission details: software@computer.org

IEEE
Software