Source-Based Software Risk Assessment

Arie van Deursen
CWI and Delft University of Technology
The Netherlands
`http://www.cwi.nl/~arie/`

Tobias Kuipers
Software Improvement Group
The Netherlands
`tobias.kuipers@software-improvers.com`

Abstract

The paper reports on a method for software risk assessments that takes into account "primary facts" and "secondary facts". Primary facts are those obtained through automatically analyzing the source code of a system, and secondary facts are those facts obtained from people working with or on the system, and available documentation. We describe how both types of facts are retrieved, and how we are bridging the interpretation gap from the raw facts (either primary or secondary) to a concise risk assessment, which includes recommendations to minimize the risk. This method has been developed while performing numerous risk assessments, and is continuously being fine-tuned.

1. Introduction

A Software Risk Assessment is an independent assessment of the risks involved in building, operating, deploying or maintaining a software system. Such audits are usually requested for a specific reason, and need to be conducted in a limited amount of time (typically less than three weeks). During the last years, we have performed such assessments both in a commercial as well as in an academic setting. We have developed a method for assessing software risks, which departs from the more traditional risk assessment approaches in that it considers the source code of the software system to be the source of so-called *primary* facts about the system. A more traditional approach to risk assessment would concern itself mostly with *process*, i.e. the way various aspects of building, operating, deploying or maintaining a software system are embedded into the organization. Specialized assessments (usually in the fields of performance or security) do sometimes consider the source code, but only with respect to a particular issue. Assessments of source code are largely done manually, as a form of code inspection, making them time consuming, error-prone, and hard to repeat. A key characteristic of our method is that it contrasts perceived problems with facts derived from the sources. This distinguishes it from source code analysis methods measuring code quality such as, for example, the DATRIX approached developed at Bell Canada [6].

Our method is very much a work in progress. The assessments we perform are done for very different reasons, requiring continuous tuning of the method. After each assessment a debriefing is done, in which the method and its application is revisited and modified to fit our goals better. This paper describes our current insights, and we explicitly solicit feedback from the academic community to improve our method.

1.1. Assessment Method

The assessment method consists of three steps, which are explained in more detail below and illustrated in Figure 1.

Secondary Fact Retrieval A system is analyzed based on information present in the organization. This information is retrieved through interviews with stakeholders, and through reviewing design documents, documentation, contracts, and so on.

Primary Fact Retrieval The source code of the system itself is analyzed. All the source code is included, which may involve software written in various languages for different subsystems, interactions with other systems, and persistent data definition and manipulation code.

Bridging the Interpretation Gap Finally, the results of the source code analysis are related to the results obtained from interviews. Since stakeholders in a software system usually have different (sometimes conflicting) views of that same system, the results obtained through code analysis help to validate those views and discern whether the risk perceived by a certain party is indeed a risk, or just a misunderstanding.

1.2. Reasons for Performing Assessments

In our experience, assessments were requested in a number of differing circumstances. Among them were the following:

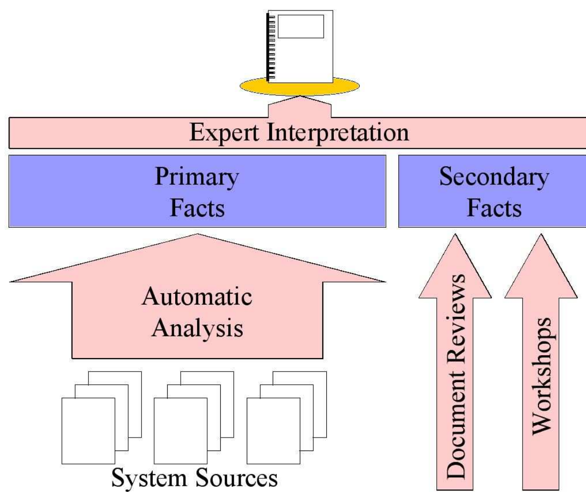

Figure 1. The steps in the assessment process

- A company has bought a standard software-package. This package does not exactly fulfill the company's requirements. The producer of the software is asked to modify the package to fit the needs of the company. After the modified package is released, the company has great difficulty implementing it, and wonders what the risk is of "going live" with the system, considering data for several million customers will be stored using this package.

- A company has bought a standard e-business package for implementing a virtual marketplace for energy trading. The maintenance of the package will be done by the company, the producer will build release 1.0, and hand over the sources. The company requests an assessment of the package, to gain insights into the maintenance risks.

- A company runs a large call-center. The software used to track the calls to the center has been outsourced for more than 15 years. In order to reduce cost, the company is planning to terminate the outsourcing contract, and maintain the software itself. An assessment is requested on the risks for the daily operation if maintenance is shifted from the outsourcer to the company.

- A government has outsourced a large administrative system a decade ago. The government feels the price it pays for the outsourcing is too high. A benchmark is performed which relates the cost of maintaining this system to the cost of maintaining similar systems. As part of this benchmark, an assessment is performed to map out the risks in the source code related to maintenance cost.

2. Secondary Fact Retrieval

Every assessment starts by organizing a workshop with the various stakeholders of the system. This workshop serves a number of purposes: First of all, it alerts everyone on the project that an assessment is going on. The workshop is used to explain the purpose of the assessment, and the assessment method. Depending on the reason for the assessment, it may be advisable to hold two workshops, for instance when there is a development team for company A, and an implementation team for company B, where A and B are not equally happy about their relationship. Indeed a major aim of the workshop is to remove fear and uncertainty. To that end, we explain that the purpose of the assessment is to identify risks and to provide recommendations to build a better system, not looking for a scapegoat.

Second, the workshop is by far the fastest way to get an enormous amount of information about the system. There usually is extensive knowledge of various specialized parts of the system. As an example we encountered, the DBA will know all about the good and bad sides of the database access, and can tell you all about how he had to configure the database parameters to values "far above the average for such a system". His knowledge of the database structure should be combined with knowledge about the source code in order to provide an accurate risk assessment.

Depending on the type of organization requesting the assessment, such information may or may not be known to the project management. Typically technical specialists will be able to see early in the project that something is out of the ordinary. The communication structure and incentives should be such that the specialists actually report this to project management — and management must be able to understand what the implication of this deviation might be.

Third, the workshop quickly focuses on what the actual questions are. Management may have the perception that a certain aspect of the system is the real problem, since it has the largest visible impact. Again an example: performance tuning the database can be as simple as adding indexes to selected columns. If this has not yet been done, the overall performance of a system may be dramatic. To management, this may seem like an enormous problem, where the database specialist knows that he can do this in a couple of hours, right after he has finished this other project that has higher priority.

The workshop contrasts various opinions about the risks (and the strong points) of the system. The target of the workshop is first of all to enumerate these opinions, possibly find a consensus, and finally make a prioritized list of the risks these opinions bring to light.

2.1. What do we bring to the workshop

Before the workshop is organized, we perform a first analysis on the source code. This gives us a rough idea on the size and architecture of the system. An important action here is to identify the so-called outliers: Modules, or subsystems that differ from the average. This can be done based simply on the number of lines of code, but also on the number of accesses

to a database, cyclomatic complexity, the number of sockets opened for reading, and so on. Here, we go with basic intuition: we have a general idea of what the system is supposed to do. We try to imagine how such a system would be designed and built, and try to match what we see with what we imagined. The imagined system serves as a first mental model of the real system: during analysis, but during the workshop too, this mental model is revised over and over again to end up resembling the actual system.

Based on past experience, and literature, we have developed a questionnaire covering a large number of topics of software system design, implementation, deployment, testing, and so on. This questionnaire is used to structure the workshop, and depending on the particular assessment, selected questions are asked to the participants of the workshop. After each assessment, the questionnaire is revised and improved.

In the spirit of SEI's Architectural Tradeoff Analysis Method ATAM [4], we have included questions concerning quality attribute tradeoffs and (future) scenario's affecting different parts of the system. Observe, however, that in this phase of the audit we are primarily trying to distill the problems as perceived by the participants: in the subsequent source code analysis phase these perceived problems are compared to the actual situation in the source code.

2.2. Who is invited to the workshop

Inviting people to the workshop can be very much a political process. Who is invited depends largely on who initiates the assessment and for what reason. However, there are a number of stakeholders that should be invited to the workshop. Depending on the political situation they may or may or be invited, and, if invited, they may or may not be allowed to join. It may be appropriate to organize two workshops, if the situation dictates it. In our experience, no more than eight people can be invited. Workshops with more than eight people tend to be hard to manage, and allow people to be inactive during the workshop.

It may be advisable to split the workshop up into an introductory part, where management is present, and a more indepth (technical) part, where management leaves the room, allowing for an uninhibited discussion of various aspects of the system. People who should generally be invited include:

Project Manager The person who is responsible for the day-to-day operation of the project, and who is ultimately responsible for the success of the project.

Customer The person who pays for the project, or if he has little or no direct involvement in the project, the most involved representative of the customer organization. In specific cases, it might be advisable to invite them both.

Architect The architect of the system, if available. This should be the person that has contributed most to the actual design of the system.

Lead developer The person who runs the daily development, and preferably does (some) development himself. Getting a middle manager with no actual knowledge of the development process is next to useless.

Lead tester The person who runs the test effort.

Various specialists Depending on the assessment, the various relevant specialists should be invited, such as database specialist, performance tuning specialist, deployment specialist, network specialist, ...

3. Primary Fact Retrieval

We have developed a framework for analyzing (very) large amounts of source code, written in different programming languages. The most prominent exponent of this framework is DocGen, a technical documentation generator permitting software system browsing at various levels of abstraction [2]. In principle, other software exploration tools could be used for such an analysis — we refer to the related work described in [7] for an overview of such tools.

The framework consists of a generic parser [1], which can be instantiated with a grammar, or language model, for a particular programming language. Source code is parsed using this parser, and then converted into Java objects using JJForester [5], a tool that generates Java classes for tree structures, such as parse trees. This results in a number of object hierarchies, representing parse trees. These can then be traversed using the visitor combinator framework JJTraveler [9]. Using JJTraveler allows us to program very sophisticated program analyses very concisely, in Java. The fact that Java is used as a development language, instead of a more academic language such as Haskell or ASF+SDF allows us to tap into the pool of readily available Java programmers.

The central part of the framework is an object model representing a number of programming language constructs. Various overlapping models exist, for procedural languages, fourth generation languages, and object-oriented languages. Where different languages of the same programming paradigm differ, the differences are modeled using inheritance. From this object model, a data model is generated, that is used to create a relational database to store the objects (using object/relational mapping).

From the database, a number of reports can be generated, either through a dedicated tool (such as the aforementioned DocGen), or through standard spreadsheet programs. This architecture allows us to customize analyses on two levels. Based on the data in the database, we can calculate specific metrics that are relevant for the assessment. We can derive charts and graphs for these metrics to make them better understandable, or to identify trends in the metrics. We can also derive lists of outliers: which module has the most database access, which has the most file operation, which calls the most

other modules. These types of analyses can be derived based on the data in the basic data model.

If the current data model does not accommodate a specific analysis, we can change the analysis on the second level: we write or specialize a new program analysis in Java to be applied on the parse tree. If necessary, the language object model should be modified to allow for the persistent storage of the results of the new analysis.

In general, modifications at the first level (after the database) are easier to perform than those at the second level (which require that all parse trees are analyzed again, a possibly time consuming operation). For specific assessments we make modifications at the second level after the workshop results make clear that a specific analysis is necessary for proper risk assessment. Modifications at the second level are never performed before the workshop; a number of standard metrics are derived, and possibly some customized metrics by creating a new analysis at the first level.

The results of the various assessments of different systems are all stored. This allows us to perform analyses on benchmarking data. For example, we can relate the results for a particular system to the results for a similar system built by a different organization. At the moment of writing the repository consists of data derived from about 45 gigabytes of source code from commercial and government organizations.

The following statistics are routinely derived: The number of modules for a system, the number lines of code, the number of programming languages, McCabe's cyclomatic complexity, the maintainability index from Oman and Hagemeister [8], the estimated number of function points using backfiring [3], the fan-in and fan-out of the modules, the number of database tables used by a module (possibly specified per database operation), the number of files used per module, the percentage of duplicated (or cloned) code (currently only for Java code), the number of parameters per module/procedure, the number of fields per database table, the number of goto statements.

4. Bridging the Interpretation Gap

The outcome of the audit is an assessment of the risks identified in the workshop based on an analysis of the primary facts retrieved from the sources. In order to be useful for management, the report should offer an *interpretation* of the observations made on the sources: it should clarify why these observations are relevant to the problems identified, and how these observations can help to find solutions for these problems.

Therefore, our reports consist of the following ingredients:

- An objective description of the problems perceived by the system stakeholders as expressed in the initial workshop;
- An objective description of source-based observations related to the problems identified;
- An objective inventory of the potential risks and benefits for each of the technical observations made. An essential role is played by benchmarks and patterns occurring in the legacy code base available.
- A subjective evaluation of these risks and benefits leading to recommendations on how to tackle the problems.

The source-based observations form the "evidence" of the report: the recommendations the best possible interpretation we can give in order to assist non-technical managers (who generally do not have the background to understand the evidence) in taking appropriate action.

5. Concluding Remarks

This paper reflects our ongoing efforts to create a systematic method for conducting a series of source-based audits. Such audits are used to take key decisions concerning future directions for existing software systems, such as replacement, integration with other systems, or outsourcing. Our audits are characterized by (i) stakeholder involvement, (ii) full system source analysis, and (iii) interpretation support. We are currently in the process of conducting additional audits, which will help us in refining and elaborating the method presented.

Acknowledgments We would like to thank the anonymous reviewers for their comments and Joost Visser for fruitful discussions.

References

[1] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation fi lters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[2] A. van Deursen and T. Kuipers. Building documentation generators. In *International Conference on Software Maintenance, ICSM'99*, pages 40–49. IEEE Computer Society, 1999.

[3] C. Jones. *Estimating Software Costs*. McGraw-Hill, 1998.

[4] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The Architecture Tradeoff Analysis Method. In *Proceedings 4th International Conference on Engineering of Complex Computer Systems*, 1998.

[5] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. *Science of Computer Programming*, 47(1):59–87, November 2002.

[6] B. Laguë. Assessing risks related to software source code using DATRIX. In *IT Procurement & Supplier Quality*, number 5, 1997.

[7] L. Moonen. *Exploring Software Systems*. PhD thesis, University of Amsterdam, December 2002.

[8] P. Oman and J. Hagemeister. Constructing and testing of polynomials predicting software maintainability. *Journal of Systems and Software*, 24(3):251–266, 1994.

[9] J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, November 2001. OOPSLA 2001 Conference Proceedings.