Chapter 1
An Introduction to Recommendation Systems in Software Engineering

Martin P. Robillard and Robert J. Walker

Abstract. Software engineering is a knowledge-intensive activity that presents many information navigation challenges. Information spaces in software engineering include the source code and change history of the software, discussion lists and forums, issue databases, component technologies and their learning resources, and the development environment. The technical nature, size, and dynamicity of these information spaces motivate the development of a special class of applications to support developers: recommendation systems in software engineering (RSSEs), which are software applications that provide information items estimated to be valuable for a software engineering task in a given context. In this introduction, we review the characteristics of information spaces in software engineering, describe the unique aspects of RSSEs, present an overview of the issues and considerations involved in creating, evaluating, and using RSSEs, and present a general outlook on the current state of research and development in the field of recommendation systems for highly-technical domains.

1.1 Introduction

Despite steady advancement in the state of the art, software development remains a challenging and knowledge-intensive activity. Mastering a programming language is no longer sufficient to ensure software development proficiency. Developers are continually introduced to new technologies, components, and ideas. The systems on which they work tend to keep growing and to depend on an ever-increasing array of external libraries and resources.

Martin P. Robillard
McGill University, Montréal, QC, Canada e-mail: martin@cs.mcgill.ca

Robert J. Walker
University of Calgary, Calgary, AB, Canada, e-mail: walker@ucalgary.ca

We have long since reached the point where the scale of the *information space*—the set of all data sources that are potentially relevant to a development project—facing a typical developer easily exceeds an individual's capacity to assimilate it. Software developers and other technical knowledge workers must now routinely spend a large fraction of their working time searching for information, for example, to understand existing code or to discover how to properly implement a feature. Often, the timely or serendipitous discovery of a critical piece of information can have a dramatic impact on productivity [6].

Although rigorous training and effective interpersonal communication can help knowledge workers orient themselves in a sea of information, these strategies are painfully limited by scale. Data mining and other knowledge inference techniques are among the ways to provide automated assistance to developers in navigating large information spaces. Just as recommendation systems for popular e-commerce websites can help expose users to interesting itemsrecommendation item previously unknown to them [15], recommendation systems can be used in technical domains to help surface previously-unknown information that can directly assist knowledge workers in their task.

Recommendation systems in software engineering (RSSEs) are now emerging to assist software developers in various activities—from reusing code to writing effective bug reports.

1.2 Information Spaces in Software Engineering

When developers join a project, they are typically faced with a *landscape* [4] of information with which they must get acquainted. Although this information landscape will vary according to organization and to the development process employed, the landscape will typically involve information from a number of sources.

The project source code. In the case of large software systems, the codebase itself will already represent a formidable information space. According to Ohloh.net, in October 2013 the source code of the Mozilla Firefox web browser totaled close to 10 million lines written in 33 different programming languages. Understanding source code, even at a much smaller scale, requires answering numerous different types of questions, such as "where is this method called?" [19]. Answering such structural questions can require a lot of navigation through the project source code [11, 17], including reading comments and identifiers, following dependencies, and abstracting details.

The project history. Much knowledge about a software project is captured in the version control system (VCS) for the project. Useful information stored in a VCS includes systematic code change patterns (e.g., files *A* and *B* were often changed together [23]), design decisions associated with specific changes (stored in commit logs), and, more indirectly, information about which developers have knowledge of which part of the code [13]. Unfortunately, the information contained in a VCS is not easily searchable or browsable. Useful knowledge must often be inferred from the VCS and other repositories, typically by using a combination of heuristics and data-mining techniques [22].

Communication archives. Forums and mailing lists, often used for informal communication among developers and other stakeholders of a project, contain a wealth of knowledge about a system [3]. Communication is also recorded in issue management systems and code review tools.

The dependent APIs and their learning resources. Most modern software development relies on reusable software assets (frameworks and libraries) exported through application programming interfaces (APIs). Like the project source code itself, APIs introduce a large, heavily-structured information space that developers must understand and navigate to complete their tasks. In addition, large and popular APIs typically come with extensive documentation [5], including reference documentation, user manuals, code examples, etc.

The development environment. The development environment for a software system includes all the development tools, scripts, and commands used to build and test the system. Such an environment can quickly become complex to the point where developers perform sub-optimally simply because they are unaware of the tools and commands at their disposal [14].

Interaction traces. It is now common practice for many software applications to collect user interaction data to improve the user experience. User interaction data consists of a log of user actions as they visit a website or use the various components of the user interface of a desktop or mobile application [8]. In software engineering, this collection of usage data takes the form of the monitoring of developer actions as they use an integrated development environment such as Eclipse [10].

Execution traces. Data collected during the execution of a software system [16, Table 3] also constitutes a source of information that can be useful to software engineers, and in particular to software quality assurance teams. This kind of dynamically-collected information includes data about the state of the system, the functions called, and the results of computation at different times in the execution of the system.

The web. Ultimately, some of the knowledge sought by or useful to developers can be found in the cloud, hosted on servers unrelated to a given software development project. For example, developers will look for code examples on the web [2], or visit the StackOverflow Questions-and-Answers (Q&A) site in the hopes of finding answers to common programming problems [12]. The problem with the cloud is that it is often difficult to assess the quality of the information found on some websites, and near impossible to estimate what information exists beyond the results of search queries.

Together, the various sources of data described above create the information space that software developers and other stakeholders of a software project will face. Although, in principle, all of this information is available to support on-going development and other engineering activities, in reality it can be dispiritingly hard to extract the answer to a specific information need from software engineering data, or in some case to even know that the answer exists. A number of aspects of software engineering data make discovering and navigating information in this domain particularly difficult.

1. The sheer amount of information available (the *information overload* problem), while not unique to software engineering, is an important factor that only grows worse with time. Automatically-collected execution traces and interaction traces, and the cumulative nature of project history data, all contribute to making this challenge more acute.
2. The information associated with a software project is *heterogeneous*, leading to a problem of high dimensionality when attempting to model or represent the data. While a vast array of traditional recommender systems can rely on the general concepts of *item* and *rating* [15], there is no equivalent universal baseline in software engineering. The information sources described above involve a great variety of information formats, including highly structured (source code), semi-structured (bug reports), and loosely structured (mailing lists, user manuals).
3. Technical information is highly *context-sensitive*. To a certain extent, most information is context-sensitive; for example, to interpret a restaurant review, it may be useful to know about the expectations and past reviews of the author. However, even in the absence of such additional context, it will still be possible to construct a coarse interpretation of the information, especially if the restaurant in question is either very good or very bad. In contrast, software engineering data can be devoid of meaning without an explicit connection to the underlying process. For example, if a large amount of changes are committed to a system's version control system on Friday afternoons, it could either mean that team members have chosen that time to merge and integrate their changes, or that a scheduled process updates the license headers at that time.
4. Software data *evolves very rapidly*. Ratings for movies can have a useful lifetime measured in decades. Restaurant and product reviews are more ephemeral, but could be expected to remain valid for at least many months. In contrast, some software data experiences high *churn*, meaning that it is modified in some cases multiple times a day [9]. For example, the Mozilla Firefox project receives around 4000 commits per month, or over 100 per day. Although not all software data gets invalidated on a daily basis (APIs can remain stable for years), the highly dynamic nature of software means that inferred facts must, in principle, continually be verified for consistency with the underlying data.
5. Software data is *partially-generated*. Many software artifacts are the result of a combination of manual and automated processes and activities, often involving a complex cycle of artifact generation with manual feedback. Examples include the writing of source code with the help of refactoring or style-checking tools, the authoring of bug reports in which the output or log of a program is copied and pasted, and the use of scripts to automatically generate mailing list messages, for example when a version of the software is released. These complex and semi-automated processes can be contrasted, for example, with the authoring of reviews by customers who have bought a certain product. In the latter

case, the process employed for generating the data is transparent, and interpreting it will be a function of the content of the item and the attributes of the author; the data generation process would not normally have to be taken into account to understand the review.

Finally, in addition to the challenging attributes of software engineering data that we noted above, we also observe that many problems in software engineering are not limited by data, but rather by computation. Consider a problem like *change impact analysis* [1, 21]: the basic need of the developer—to determine the impact of a proposed change—is clear, but in general it is impossible to compute a precise solution. Thus, in software engineering and other technical domains, guidance in the form of recommendations is not only needed to navigate large information spaces, but also to deal with *formally undecidable problems*, or problems where no precise solutions can be computed in a practical amount of time.

1.3 Recommendation Systems in Software Engineering

In our initial publication on the topic, we defined a recommendation system for software engineering to be [18, p.81]:

> ... a software application that provides information items estimated to be valuable for a software engineering task in a given context.

With the perspective of an additional four years, we still find this definition to be the most useful for distinguishing RSSEs from other software engineering tools. RSSEs' focus is on providing *information* as opposed to other services such as build or test automation. The reference to *estimation* distinguishes RSSEs from fact extractors, such as classical search tools based on regular expressions or the typical cross-reference tools and call-graph browsers found in modern integrated development environments. At the same time, estimation is not necessarily *prediction*: recommendation systems in software engineering need not rely on the accurate prediction of developer behavior or system behavior. The notion of *value* captures two distinct aspects simultaneously: (1) novelty and surprise, because RSSEs support discovering new information; and (2) familiarity and reinforcement, because RSSEs support the confirmation of existing knowledge. Finally, the reference to a specific *task* and *context* distinguish RSSEs from generic search tools, e.g., tools to help developers find code examples.

Our definition of RSSEs is, however, still broad and allows for great variety in recommendation support for developers. Specifically, a large number of different information items can be recommended, including:

Source code within a project. Recommenders can help developers navigate the source code of their own project, for example by attempting to guess the areas of the project's source code a developer might need, or want, to look at.

Reusable source code. Other recommenders in software engineering attempt to help users discover the API elements (such as classes, functions, or scripts) that can help to complete a task.

Code examples. In some cases, a developer may know which source code or API elements are required to complete a task, but may ignore how to correctly employ them. As a complement to reading textual documentation, recommendation systems can also provide code examples that illustrate the use of the code elements of interest.

Issue reports. Much knowledge about a software project can reside in its issue database. When working on a piece of code or attempting to solve a problem, recommendation systems can discover related issue reports.

Tools, commands, and operations. Large software development environments are getting increasingly complex, and the number of open-source software development tools and plug-ins is unbounded. Recommendation systems can help developers and other software engineers by recommending tools, commands, and actions that should solve their problem or increase their efficiency.

People. In some situations recommendation systems can also help finding the best person to assign a task to, or the expert to contact to answer a question.

Although dozens of RSSEs have been built to provide some of the recommendation functionality described above, no reference architecture has emerged to-date. The variety in RSSE architectures is likely a consequence of the fact that most RSSEs work with a dominant source of data, and are therefore engineered to closely integrate with that data source. Nevertheless, the major design concerns for recommendation systems in general are also found in the software engineering domain, each with its particular challenges.

Data preprocessing. In software engineering, a lot of preprocessing effort is required to turn raw character data into a sufficiently interpreted format. For example, source code has to be parsed, commits have to be aggregated, software has to be abstracted into dependency graphs, etc. This effort is usually needed in addition to more traditional preprocessing tasks such as detecting outliers and replacing missing values.

Capturing context. While in traditional domains, such as e-commerce, recommendations are heavily dependent on user profiles, in software engineering, it is usually the *task* that is the central concept related to recommendations. The *task context* is our representation of all information about the task to which the recommendation system has access in order to produce recommendations. In many cases, a task context will consist of a partial view of the solution to the task: for example, some source code that a developer has written; an element in the code that a user has selected; or an issue report that a user is reading. Context can also be specified explicitly, in which case the definition of the context becomes fused with that of a query in a traditional information retrieval system. In any case, capturing the context of a task to produce recommendations involves somewhat of a paradox: the more precise the information available about the task is, the more accurate the recommendations can be, but the less likely the user can expected to need recommendations. Put another way, a user in great need of guidance may not be able to provide enough information to the system to obtain usable recommendations. For this reason, recommendation systems must take into account that task contexts will generally be incomplete and noisy.

Producing recommendations. Once preprocessed data and a sufficient amount of task context are available, recommendation algorithms can be executed. Here the variety of recommendation strategies is only bounded by the problem space and the creativity of the system designer. However, we note that the traditional recommendation algorithms commonly known as collaborative filtering are only seldom used to produce recommendations in software engineering.

Presenting the recommendations. In its simplest form, presenting a recommendation boils down to listing items of potential interest—functions, classes, code examples, issue reports, and so on. Related to the issue of presentation, however, lies the related question of *explanation*: why was an item recommended? The answer to this question is often a summary of the recommendation strategy: "average rating", "customers who bought this item also bought", etc. In software engineering, the conceptual distance between a recommendation algorithm and the domain familiar to the user is often much larger than in other domains. For example, if a code example is recommended to a user because it matches part of the user's current working code, how can this matching be summarized? The absence of a universal concept such as ratings means that for each new type of recommendation, the question of explanation must be revisited.

1.4 Overview of the Book

In the last decade, research and development on recommendation systems has seen important advances, and the knowledge relevant to recommendation systems now easily exceeds the scope of a single book. This book focuses on the development of recommendations systems for technical domains and, in particular, for software engineering. The topic of recommendation systems in software engineering is broad to the point of multidisciplinarity: it requires background in software engineering, data mining and artificial intelligence, knowledge modeling, text analysis and information retrieval, human–computer interaction, as well as a firm grounding in empirical research methods. This book was designed to present a self-contained overview that includes sufficient background in all of the relevant areas to allow readers to quickly get up to speed on the most recent developments, and to actively use the knowledge provided here to build or improve systems that can take advantage of large information spaces that include technical content.

Part I of the book covers the foundational aspects of the field. Chapter 2 presents an overview of the general field of recommendation systems, including a presentation of the major classes of recommendation approaches: collaborative filtering, content-based recommendations, and knowledge-based recommendations. Many recommendation systems rely on data mining algorithms; to help readers orient themselves in the space of techniques available to infer facts from large data sets, Chap. 3 presents a tutorial on popular data mining techniques. In contrast, Chap. 4 examines how recommendation systems can be built without data mining, by relying instead on carefully designed heuristics. To-date, the majority of RSSEs have targeted the recommendation of source code artifacts; Chap. 5 is an extensive review of recommendation systems based on source code that includes many examples of RSSEs. Moving beyond source code, we examine two other important sources of data for RSSE: bug reports in Chap. 6, and user interaction data in Chap. 7. We conclude Part I with two chapters on human–computer interaction (HCI) topics: the use of developer profiles to take personal characteristics into account, in Chap. 8, and the design of user interfaces for delivering recommendations, in Chap. 9.

Now that the field of recommendation systems has matured, many of the basic ideas have been tested, and further progress will require careful, well-designed evaluations. Part II of the book is dedicated to the evaluation of RSSEs with four chapters on the topic. Chapter 10 is a review of the most important dimensions and metrics for evaluating recommendation systems. Chapter 11 focuses on the problem of creating quality benchmarks for evaluating recommendation systems. The last two chapters of Part II describe two particularly useful types of studies for evaluating RSSEs: simulation studies that involve the execution of the RSSE (or of some of its components) in a synthetic environment (Chap. 12), and field studies, which involve the development and deployment of an RSSE in a production setting (Chap. 13).

Part III of the book takes a detailed look at a number of specific applications of recommendation technology in software engineering. By discussing RSSEs in an end-to-end fashion, the chapters in Part III provide not only a discussion of the major concerns and design decisions involved in developing recommendation technology in software engineering, but also provide insightful illustrations of how computation can assist humans in solving a wide variety of complex, information-intensive tasks. Chapter 14 discusses the techniques underlying the recommendation of reusable source code elements. Chapters 15 and 16 present two different approaches to recommend transformations to an existing code base. Chapter 17 discusses how recommendation technology can assist requirements engineering, and Chap. 18 focuses on recommendations that can assist tasks involving issue reports, such as issue triage tasks. Finally, Chap. 19 shows how recommendations can assist with product line configuration tasks.

1.5 Outlook

As the content of this book shows, the field of recommendation systems in software engineering has already benefited from much effort and attention from researchers, tool developers, and organizations interested in leveraging large collections of software artifacts to improve software engineering productivity. We conclude this introduction with a look at the current state of the field and the road ahead.

Most of the work on RSSEs to-date has focused on the development of algorithms for processing software data. Much of this work has proceeded in the context of the rapid progress in techniques to mine software repositories. As a result, developers of recommendation systems in software engineering can now rely on a mature body of knowledge on the automated extraction and interpretation of software data [7]. At the same time, developments in RSSEs had, up to recently, proceeded somewhat in isolation of the work on traditional recommender systems. However, the parallel has now been recognized, which we hope will lead to a rapid convergence in terminology and concepts that should facilitate further exchange of ideas between the two communities.

Although many of the RSSEs mentioned in this book have been fully implemented, much less energy has been devoted to research on the human aspects of RSSEs. For a given RSSE, simulating the operation of a recommendation algorithm can allow us to record very exactly how the algorithm would behave in a large number of contexts, but provides no clue as to how users would react to the recommendations (see Part II). For this purpose, only user studies can really provide an answer. The dearth of user studies involving recommendation systems in software engineering can be explained and justified by their high cost, which would not always be in proportion to the importance of the research questions involved. However, the consequence is that we still know relatively little about how to best integrate recommendations into a developer's workflow, how to integrate recommendations from multiple sources, and more generally how to maximize the usefulness of recommendation systems in software engineering.

An important distinction between RSSEs and traditional recommendation systems is that RSSEs are task-centric, as opposed to user-centric. In many recommendation situations, we know much more about the task than about the developer carrying it out. This situation is reflected in the limited amount of personalization in RSSEs. It remains an open question whether personalization is necessary or even desirable in software engineering. As in many cases, the accumulation of personal information into a user (or developer) profile has important privacy implications. In software engineering, the most obvious one is that this information could be directly used to evaluate developers. A potential development that could lead to more personalization in recommender systems for software engineering is the increasingly pervasive use of social networking in technical domains. Github is already a platform where the personal characteristics of users can be used to navigate information. In this scenario, we would see a further convergence between RSSE and traditional recommenders.

Traditional recommendation systems provide a variety of *functions* [15, Sect. 1.2]. Besides assisting the user in a number of ways, these functions also include a number of benefits to other stakeholders, including commercial organizations. For example, recommendation systems can help increase the number of items sold, sell more diverse items, increase customer loyalty, etc. Although, in the case of RSSEs developed by commercial organizations, these functions can be assumed, we are not aware of any research that focuses on assessing the non-technical virtues of RSSEs. At this point, most of the work on assessing RSSEs has focused on the support they directly provide to developers.

1.6 Conclusion

The information spaces encountered in software engineering contexts differ markedly from those in non-technical domains. Five aspects—quantity, heterogeneity, context-sensitivity, dynamicity, and partial generation—all contribute to making it especially difficult to analyze, interpret, and assess the quality of software engineering data. The computational intractability of many questions that surface in software engineering only add to the complexity. Those are the challenges facing organizations that wish to leverage their software data.

Recommendation systems in software engineering are one way to cope with these challenges. At heart, they must be designed to acknowledge the realities of the tasks, of the people, and of the organizations involved. And while RSSEs give rise to new challenges, we have already learned a great deal about techniques to create them, methodologies to evaluate them, and details of their application.

References

1. Arnold, R.S., Bohner, S.A.: Impact analysis: Towards a framework for comparison. In: Proceedings of the Conference on Software Maintenance, pp. 292–301 (1993). DOI 10.1109/ICSM.1993.366933
2. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., Klemmer, S.R.: Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 1589–1598 (2009). DOI 10.1145/1518701.1518944
3. Čubranić, D., Murphy, G.C., Singer, J., Booth, K.S.: Hipikat: A project memory for software development. IEEE Transactions on Software Engineering **31**(6), 446–465 (2005). DOI 10.1109/TSE.2005.71
4. Dagenais, B., Ossher, H., Bellamy, R.K., Robillard, M.P.: Moving into a new software project landscape. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 275–284 (2010)
5. Dagenais, B., Robillard, M.P.: Creating and evolving developer documentation: Understanding the decisions of open source contributors. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 127–136 (2010). DOI 10.1145/1882291.1882312
6. Duala-Ekoko, E., Robillard, M.P.: Asking and answering questions about unfamiliar APIs: An exploratory study. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 266–276 (2012)
7. Hemmati, H., Nadi, S., Baysal, O., Kononenko, O., Wang, W., Holmes, R., Godfrey, M.W.: The MSR cookbook: Mining a decade of research. In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 343–352 (2013). DOI 10.1109/MSR.2013.6624048

8. Hill, W.C., Hollan, J.D., Wroblewski, D.A., McCandless, T.: Edit wear and read wear. In: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 3–9 (1992). DOI 10.1145/142750.142751
9. Holmes, R., Walker, R.J.: Customized awareness: Recommending relevant external change events. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 465–474 (2010). DOI 10.1145/1806799.1806867
10. Kersten, M., Murphy, G.C.: Mylar: A degree-of-interest model for IDEs. In: Proceedings of the International Conference on Aspect-Oriented Software Deveopment, pp. 159–168 (2005). DOI 10.1145/1052898.1052912
11. Ko, A.J., Myers, B.A., Coblenz, M.J., Aung, H.H.: An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Transactions on Software Engineering **32**(12), 971–987 (2006). DOI 10.1109/TSE.2006.116
12. Kononenko, O., Dietrich, D., Sharma, R., Holmes, R.: Automatically locating relevant programming help online. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 127–134 (2012). DOI 10.1109/VLHCC.2012.6344497
13. Mockus, A., Herbsleb, J.D.: Expertise Browser: A quantitative approach to identifying expertise. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 503–512 (2002). DOI 10.1145/581339.581401
14. Murphy-Hill, E., Jiresal, R., Murphy, G.C.: Improving software developers' fluency by recommending development environment commands. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 42:1–42:11 (2012). DOI 10.1145/2393596.2393645
15. Ricci, F., Rokach, L., Shapira, B.: Introduction to Recommender Systems Handbook. In: Ricci, F., Rokach, L., Shapira, B. (eds.) Recommender Systems Handbook, pp. 1–35. Springer (2011). DOI 10.1007/978-0-387-85820-3_1
16. Robillard, M.P., Bodden, E., Kawrykow, D., Mezini, M., Ratchford, T.: Automated API property inference techniques. IEEE Transactions on Software Engineering **39**(5), 613–637 (2013). DOI 10.1109/TSE.2012.63
17. Robillard, M.P., Coelho, W., Murphy, G.C.: How effective developers investigate source code: An exploratory study. IEEE Transactions on Software Engineering **30**(12), 889–903 (2004). DOI 10.1109/TSE.2004.101
18. Robillard, M.P., Walker, R.J., Zimmermann, T.: Recommendation systems for software engineering. IEEE Software **27**(4), 80–86 (2010). DOI 10.1109/MS.2009.161
19. Sillito, J., Murphy, G.C., De Volder, K.: Asking and answering questions during a programming change task. IEEE Transactions on Software Engineering **34**(4), 434–451 (2008). DOI 10.1109/TSE.2008.26
20. Walker, R.J., Holmes, R., Hedgeland, I., Kapur, P., Smith, A.: A lightweight approach to technical risk estimation via probabilistic impact analysis. In: Proceedings of the International Workshop on Mining Software Repositories, pp. 98–104 (2006). DOI 10.1145/1137983.1138008
21. Weiser, M.: Program slicing. IEEE Transactions on Software Engineering **10**(4), 352–357 (1984). DOI 10.1109/TSE.1984.5010248
22. Zimmermann, T., Weißgerber, P.: Preprocessing CVS data for fine-grained analysis. In: Proceedings of the International Workshop on Mining Software Repositories, pp. 2–6 (2004)
23. Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. IEEE Transactions on Software Engineering **31**(6), 429–445 (2005). DOI 10.1109/TSE.2005.72