Part II. Evaluation: Field Studies

Book Section

Field Studies in the Construction and Evaluation of Recommendation Systems in Software Engineering

Ayşe Tosun Mısırlı, Ayşe Bener, Bora Çağlayan, Gül Çalıklı and Burak Turhan

Abstract One way to implement and evaluate the effectiveness of recommendation systems for software engineering is to conduct field studies. Field studies are important as they are the extension of the laboratory experiments into real life situations of organizations and/or society. They bring greater realism to the phenomenon that is under study. However, field studies require following a rigorous research approach with many challenges attached, such as difficulties in implementing the research design, achieving sufficient control, replication, validity and reliability. In practice, another challenge is to find organizations who are prepared to be experimented on. In this chapter, we provide details regarding step-by-step process in the construction and deployment of recommendation systems for software engineering in the field. We also emphasize three main challenges (organizational, data, design) encountered during field studies in both general and specific to software organizations.

1 What is a Field Study?

Field studies have been a well established research methodology in social sciences for a long time. Field study is defined as a study that takes place in natural envi-

Ayşe Tosun Mısırlı
University of Oulu, 90014, Oulu, Finland e-mail: ayse.tosunmisirli@oulu.fi

Ayşe Bener
Ryerson University, Toronto, ON, Canada e-mail: ayse.bener@ryerson.ca

Bora Çağlayan
Boğaziçi University, 34342, Bebek, Istanbul, Turkey e-mail: bora.caglayan@boun.edu.tr

Gül Çalıklı
Ryerson University, Toronto, ON, Canada e-mail: gul.calikli@ryerson.ca

Burak Turhan
University of Oulu, 90014, Oulu, Finland e-mail: burak.turhan@oulu.fi

ronment of the subject of the study rather than in a laboratory environment, and it involves observations, experiments, and interactions with participants. Field studies focus on what really matters for practice in setting the research agenda [22]. In this sense, field studies are critical to understand the problems in practice to come up with solutions that can be turned into action by the practitioners.

In this section, we initially present different philosophical views, research methodologies, data collection and analysis methods that must be determined at the beginning of a field study. Later, we provide a recipe for readers who are interested in building and deploying a recommendation system for software engineering by explaining the main steps and considerations.

1.1 Understanding the problem in the field

Next generation of empirical research in software engineering needs to focus more on producing actionable outcomes. Software development is one of the domains that complex interactions among the product, the process that enables the creation of the product and the people who create the product and the processes take place. In this complex environment, there are many uncertainties and blind spots that need to be covered. Software continuously changes, and as it changes, it becomes increasingly unstructured. Software engineering is hard, merely because the product is flexible, intangible and complex. On the other hand, software engineering is one of the rare fields where academic research overlaps with the needs of industry. Depending on the research question, a field study in a software organization may involve observing people or processes, reviewing code or documentation, collecting metrics, interviewing people, and making a qualitative, quantitative or mixed analysis.

Software engineering is challenging by its nature, and it is the very same nature that makes software engineering a data rich domain in terms of the artefacts and decisions made daily. Therefore, there are already many recommendation systems to assist software professionals in various activities, and new ones keep emerging to address the needs arising from ever evolving, complex and heterogeneous nature of software systems [37]. Recommendation systems in software organizations are built to support decision making under uncertainty and to ease the delivery of a task or a process. The output of a recommendation system is used by all levels of a development team to overcome technical challenges, i.e., to efficiently manage/organize technical resources, to improve the quality of their work, to find and solve problems in the process, or the code itself. As recommendation systems mature, they eventually become the corporate history with their capabilities of simple and complex data analysis techniques. In this sense, recommendation systems for software engineering are good examples of field studies in this domain.

Conducting field studies are advantageous both for researchers and software practitioners. With field studies, researchers are able to understand the domain, main challenges for practitioners during software development, collect real data and propose simple solutions through different types of recommendation systems depend-

ing on the problem. Practitioners, on the other hand, can benefit from field studies, more specifically from the outputs of field studies in the form of recommendation systems, measurement repositories, etc. These systems can address common (e.g. code completion [4]), as well as specific problems (e.g. prediction of code pieces that require performance improvement [19]) faced by software practitioners in the field, and guide them in cases where the experience of the user is insufficient to make a justified decision, or the required data processing is intractable/ infeasible in terms of time and resources.

In general, benefits of recommendation systems in software engineering field is time and effort reduction for accomplishing the task at hand. Depending on the specific problem, this saving can vary from minutes (e.g. API Discoverability in [14], Debug Advisor in [2]) to months (e.g. Performance debugging in [19]) scale. In addition to the direct benefit of time and effort reduction, recommendation systems also have indirect benefits during field usage, e.g. increased developer productivity [4], faster development and maintenance cycles [14, 19], improved bug fixing [2], more accurate fault localization [3], and reduced workload and information overload [21].

Designing and conducting a field study need to follow a consistent process. In the next subsections, we will briefly explain this process step-by-step for researchers who would like to initiate a field study in a software organization, as well as for practitioners who are the main participants of a field study in deciding the research methodology and data collection methods.

1.1.1 Research Philosophies

The research process involves deciding on which methodologies and methods to be employed and how a researcher would justify the choice and use of the methodologies and methods. Crotty proposes a framework that combines methods, methodologies, theoretical perspectives and epistemologies [10]. Method is defined as the techniques or procedures used to gather and analyse data related to research questions or hypothesis. Methodology is defined as the strategy, plan of action process or design behind the choice and use of particular methods and linking the choice and use of methods to the desired outcomes. Theoretical perspective is the philosophical stance informing the methodology and thus providing a context for the process and grounding its logic and criteria. Epistemology is the theory of knowledge embedded in theoretical perspective and thereby the methodology. In other words, epistemology helps the researcher to determine his/ her theoretical perspective, then the theoretical perspective determines the methodology, and then the methodology follows the usage of methods that are consistent with the methodology.

Some researchers refer to epistemology by using different terminologies. Creswell calls it “worldviews”, meaning “a basic set of beliefs that guide action” ([9], p.6; [16], p.17). Others call it “research paradigms” [25, 30], and “broadly conceived research methodologies” [35]. Table 1.1.1 presents different classifications of world-

views, which is adopted from Creswell [9], and modified with examples from software engineering for different research designs.

Before any empirical research begins, a researcher needs to decide what his/ her worldview is depending on the nature of the research question they have. Then the researcher will decide which methodology to employ that is consistent with the worldview he/ she has. Next the research methods that is consistent with the methodology chosen will be determined. In empirical software engineering research, for example, if a constructivist approach is taken, researcher formulates a hypothesis or question to test, then observes the situation, abstracts observations into data, then analyses the data and draws conclusions with respect to tested hypothesis. In building a recommendation system for software engineering, if we take a qualitative approach we would observe how software teams work, understand the processes, collect data through interviews, interpret the data we gathered and then build a recommendation system to help practitioners determine the project cost or understand the reasons for a defect/issue so that the recommendation system creates an agenda for change in the process or team structure or both.

Research Designs	**Qualitative Approaches**	**Quantitative Approaches**	**Mixed Method Approaches**
Philosophical assumptions	Constructivist/ Advocacy/ Participatory	Post-positivist	Pragmatic
The strategies of inquiry	Phenomenology, grounded theory, ethnography, case study & narrative	Surveys & experiments	Sequential, concurrent, & transformative
Methods	Open-ended questions, emerging approaches, text or image data	Closed-ended questions, predetermined approaches, numeric data	Both open- and closed-ended questions, both emerging and predetermined approaches, & both quantitative and qualitative data and analysis
Practices of Research	• Position him- or herself • Collect participant meanings • Bring personal values into the study • Study the context or setting of participants • Validate the accuracy of findings • Make interpretations of the data • Create an agenda for change or reform • Collaborate with the participants	• Test and verify theories or explanations • Identify variables to study • Relate variables in questions or hypotheses • Uses standards of validity and reliability • Observe and measures information numerically • Use unbiased approaches • Employ statistical procedures	• Collect both quantitative and qualitative data • Develop a rationale for mixing • Integrate the data at different stages of inquiry • Present visual pictures of the procedures in the study • Employ the practices of both qualitative and quantitative research
Example implementations in software engineering research	Cost/Development effort estimation models, Rule-based models	Predictive models using regression analysis or data mining	Hybrid Bayesian models, Case-based reasoning

Table 1 Research Designs as World-views, Strategies and Methods, adopted from [9]

1.1.2 Research Methodologies

Research methodologies or strategies of inquiry can be qualitative, quantitative or the combination of the two [9].

In quantitative research there is a controlled setting for research, and the subject is an object of the research. Research designed is fixed and the researcher is outside of the setting. The emphasis is on reliability and generalizability.

In qualitative research there is complex and real world setting for the research. Subject is participant in the research process. Research design evolves and the researcher is inside the setting. The emphasis is on validity of the research. There are different types of qualitative research strategies as shown in Table 1.1.1.

Field studies in software engineering in the context of building recommendation systems should consider employing more qualitative research approaches to gain insights and sufficiently reflect domain knowledge. Qualitative approaches such as case studies would help building local models so that we can use quantitative approaches (i.e. statistics, data mining) to generalize for other settings.

1.1.3 Research Methods

Research methods involve data collection, analysis, interpretation and presentation of results. Depending on the methodology chosen by the researcher, research method will differ. Quantitative methods require that empirical investigation is based on statistical, mathematical models, and computational techniques. Qualitative methods, on the other hand, are based on observations, interviews, surveys with open ended questions to find themes and patterns for interpretation.

In summary, selecting a particular research design is based on the research problem. If the problem calls for explanation or theory testing, then quantitative approach is appropriate. If the problem calls for exploration and understanding of the phenomena, then qualitative approach is appropriate. If the problem is such that one approach alone would not be sufficient to address, then mixed methods are appropriate [9]. Field studies in the context of recommendation systems may employ either research design. In the literature, both quantitative and qualitative research design is employed in building a recommendation system. In most cases, a recommendation system takes quantitative and qualitative data, uses data mining techniques and predictive analytics to make a recommendation and then, the system is further calibrated with expert feedback (qualitative data again) such that the recommender system and the expert (developer, tester, manager, etc.) work in collaboration to better understand and deal with the complexities of current systems [39].

1.2 Conducting Field Studies: Step by Step Design Process

A recommendation system typically goes through several iterations of five steps in a field study: Planning and negotiation, data collection and analysis, technology development (initial prototype of the recommendation system), calibration and deployment. In this section, we describe these steps and how each step evolves in a field study by giving real examples from the literature. Since there is an active collaboration between researchers and practitioners in a field study, we define technology development as a combined activity with calibration, under "building the recommendation system". Table 1.2 summarizes five main phases for building recommendation systems in software engineering, and important tasks that need to be considered for each phase in conducting field studies.

Phase	**Main considerations**
I: Planning and negotiation	- Identify business goals, challenges during software development - Align objectives of field study with practitioners' expectations - Define roles and responsibilities of parties - Initial negotiation between researchers and practitioners on the input, functionality, output of a recommendation system
II: Data collection & analysis	- Select data collection technique(s): Direct, indirect, independent, or mixed - Select data analysis technique(s) depending on the selection above: Quantitative, qualitative, or mixed - Involvement of software practitioners into the discussion of data accuracy and analysis
III & IV: Building the recommendation system and local calibration	- Design the system in terms of input, recommendation engine, output - Implement the recommendation engine - Select type of experiment(s) for performance evaluation: Offline, user stories, online
V: Deployment of the recommendation system	- Integrate with existing systems: Standalone or plug-in - Improve the system continuously through user feedback: Post-usage calibration

Table 2 Main steps in construction and evaluation of recommendation systems in software engineering

1.2.1 Phase I: Planning and negotiation

Field studies are particularly useful in the context of recommendation systems, since the aim is to propose a practical solution to a real challenge in software organizations, and ultimately, to deploy this solution as a tool into existing systems. To accomplish this, initial steps of a field study should be identifying business goals,

long-term and short-term strategies and the challenges in a software organization. Then, these business goals should be mapped to research questions of researchers explicitly.

During this step, direct interaction with software practitioners is very essential. Interactions through brainstorming sessions, formal or informal interviews, and observations of existing development environments that show patterns of activities, goals and rationales are valid to understand the needs of practitioners.

In a longitudinal field study conducted by Tosun et al. [47], a defect prediction model was built for a large scale software organization in order to improve software quality by effectively allocating testing resources. Authors presented that they made kick-off meetings with the senior management, project leads and members of development team to identify business needs and challenges in development life cycle. Research goals of the field study were also aligned with management expectations and the roles and responsibilities of both parties (researchers and practitioners) were clearly defined.

In [32], authors conducted a field study in a large software development organization, where they built a recommendation system to locate people with desired expertise throughout large parts of software product. At the beginning, interviews were done with geographically distributed development groups in order to learn the existing problem and expectations from a recommendation system. Based on the interviews, authors realized that the existing techniques for finding experts are highly uncertain and time consuming and they imposed a burden on certain individuals (Often, they were software architects). So, a simple recommender system that locates the obvious expert was not desired. After initial interviews, the objective of this recommendation system (*Expertise Browser*) was set as escalating new developers apart from architects and allowing users to find expertise through guidance.

After identifying the problem, negotiation with practitioners should be made regarding the input required to feed the recommender system, functionalities of the system, the output given as a recommendation, and a detailed planning, i.e., the time that will be spent for data collection, analysis and deployment. Field studies require frequent communication between researchers and practitioners in every step, such as "data collection, building the recommendation engine, and reporting with an interface". To avoid disappointment or dissatisfaction about the final output, initial project planning meetings should be taken seriously.

1.2.2 Phase II: Data collection & analysis

Data collection strategies: There are different data collection strategies based on research questions defined before conducting a field study. Singer et al. [40] extensively discussed a series of data collection techniques for conducting a field study and their advantages/disadvantages (see Chapter 1). According to authors' taxonomy, there are *direct*, *indirect* and *independent* data collection techniques which differ from each other in terms of how much contact is required between researchers and practitioners.

Generally, direct techniques require more resources (e.g. interviews, brainstorming sessions [40]), since they involve the highest amount of interaction with the practitioners, but the volume of data collected with these techniques is often small to medium. Indirect techniques, on the other hand, help to gather data from instrumenting systems that development team actively uses (e.g. how frequently a tool is used, timing underlying different activities, commands developers write if available) and they build an indirect relation between practitioners and researchers. These techniques require very little time from software practitioners and hence, they are appropriate for longitudinal studies.

There are also *independent* techniques which aim to collect the output and by-products of development without intervention of software practitioners. Example outputs are source codes, documentations, whereas by-products are work requests, change logs, build and configuration management tools (data repositories). Data collected through independent techniques can be then transformed to inputs of recommendation systems.

Recommendation systems in software engineering (RSSE) have become more popular with the usage of publicly available data, such as source code and other repositories, often collected with indirect and independent techniques [37]. Direct techniques, such as interviews and surveys with practitioners, are also preferred during evaluation of prototypes of recommendation systems. With the adoption of common software development interfaces, such as Bugzilla for issue management and Eclipse as a development and tool integration platform, it has become easier to adopt indirect techniques for data collection in field studies. For example, Eclipse *Metrics* plug-in can be used to gather object oriented metrics from the source code and make recommendations on product quality [15]). Another Eclipse plug-in, *CPD* (or Copy Paste detector), automatically collects copy-paste patterns from the source code to make recommendations on duplicated code [8]. In [37], authors gave other examples for different recommendation systems with examples of independent techniques, i.e., embedded plug-ins, for data collection. For more information about mining software repositories, readers can check the proceedings of MSR conference (http://2013.msrconf.org/history.php).

There are also stand alone applications as all-in-one systems including both independent and indirect data collection techniques, analysis and recommendation engines. Caglayan et al. [5] proposed *Dione* as a stand alone metric extraction and recommendation tool. Dione collects raw data by connecting to source code and issue repositories in a software organization, transforms them into metrics representing software artefacts including process, product, people, and it recommends list of defect-prone modules in the source code.

Other RSSE such as locating people with desired expertise in a particular area of the code (*Expertise Browser* [32]) uses a mixed data collection approach. The tool automatically collects data from change management systems. Additionally, authors also conducted interviews (one of the direct techniques) with practitioners to evaluate whether the expertise recommended by the tool is aligned with the expertise assigned to people by their peers. Similarly, Anvik and Murphy [1] built a recommender system to reduce bug triage efforts in a field study and evaluated the

performance by using direct data collection techniques, such as questionnaires and post-usage interviews.

Data Analysis Techniques: There are book chapters (see [53], [40]) and journal papers (see [24]) explaining qualitative and quantitative data analysis techniques that are widely used in conducting any type of empirical software engineering research. Seaman explained qualitative data analysis for exploratory, confirmatory or grounded theory studies and for visualization (Chapter 2 in [40]), whereas Rosenberg defined the measurement process of quantitative data and alternative data analysis techniques for description, comparison or prediction (Chapter 6 in [40]). Wohlin et al. [53], on the other hand, focused on quantitative data analysis in three steps of a controlled experiment: descriptive statistics, reducing the data set (i.e., data cleaning/processing) and hypothesis testing.

In field studies, depending on research questions and data collection methodology, researchers should choose data analysis techniques that are best suited. If *direct* techniques are used in the form of interviews, focus groups, surveys, etc., rigorous analysis is necessary and it often requires more time and effort than expected (e.g. coding all responses in a survey to measurable units). In this case, researchers also have the possibility to perform data analysis in parallel with data collection as soon as significant amount of data has been collected [40]. However, if *indirect* or *independent* techniques are chosen, depending on research questions and hypotheses formed at the beginning of research, quantitative analysis techniques, such as appropriate statistical tests (considering distributional assumptions of data) and data cleaning algorithms (noise detection or removal) should be used [53].

As a final point, in the context of recommendation systems, it is generally more important and useful to discuss the results of data analysis with practitioners in the field. Practitioners can tell researchers whether analysis results are accurate portrayals of the existing situation or whether they are potential outliers or noise in the original data. This also helps researchers choose appropriate analysis techniques and algorithms when building the systems.

For more detailed explanation of data analysis, we suggest readers to read Yin's book on case study research [55], or [53, 40] to see quantitative versus qualitative techniques in software engineering, or [24] to choose a data collection method in field studies and decide the best analysis techniques by considering the advantages and disadvantages of the selected method.

1.2.3 Phases III & IV: Building the recommendation system and local calibration

Before building a recommendation system in the field, researchers should rigorously work on the design in terms of input provided to the system, output that will be produced, and the engine which is the core part of the system calculating the recommendations. In this subsection, we will define these dimensions and give examples about their implementation from recommendation systems in the field.

Robillard et al. [37] defines input and output dimensions as follows:

- Input is the context of recommendation system that is provided by the user explicitly or extracted from system implicitly.
- Output is the recommendation that will be produced by the system, and it can have two types: Pull mode, i.e., producing the output when it is requested by the developer (run only when the user asks for it), Push mode, i.e., producing continuous output (continuous and up-to-date feedback as new data comes).

In a field study, input and output modes are often specified through initial negotiations with practitioners, i.e., based on real users' specific needs, how much novelty or risk they are seeking, and in what granularity or frequency they want to get recommendations. However, these can be rough estimates at Phase I, or subject to change during phase IV (calibration) based on practitioners' experimentation with the system and opinions after usage.

Recommendation engine, on the other hand, is the real intelligence provided by researchers during Phase III. This dimension mainly consists of data gathering and analysis, implementation of a model with predictive power, and selection of a ranking algorithm that systematically lists recommendations from the most valuable to the least. According to Shani and Gunawardana [38], the engine can be tested in several ways: *Offline, user studies, online*.

- *Offline:* This type of experimentation is done without user interaction, using existing data sets from public data repositories, open source systems or using a pre-collected data set from a software organization. The accuracy of the system, i.e., prediction accuracy like defect detection capability in [47], is initially evaluated during offline experiments. In doing so, researchers assume that data available is similar enough to what will be provided after the system is deployed. Offline experiments are useful to test a set of candidate algorithms with a lower cost compared to the other ways of experimentation.
- *User studies:* These are conducted by selecting a small group of subjects to use the system or a pilot project (component) among many projects (components of a large project) in a software organization. While users perform their tasks, researchers may record users' behaviours, collect any quantitative statistics and gather feedback after the usage [38].
  For example, in [14], a recommendation system called APIExplorer was evaluated through a pilot study with 32 sessions of participants. APIExplorer is developed as a feature in Eclipse IDE that discovers and recommends methods of types in APIs that are not directly reachable from the types developers are currently working with. During this pilot usage, real life APIs were selected to see how the model would perform if it were deployed, and it was calibrated with both algorithm refinements and user feedback.
  Another example for user studies can be found in [47], where a defect prediction model was executed on a group of projects in a large software system, and based on its prediction accuracy, i.e. high false alarm rates, it was calibrated by adding new information (churn metrics) from the repositories before deployment.
  A similar methodology is applied by Guo et al. in [18], in which an automated recommendation system detecting code smells (symptoms of poor programming

practices) in the source code was proposed by tailoring code smell definitions based on domain specific information. Authors conducted a focus group study with an industrial partner which includes 3 professionals from the maintenance team and asked them in a panel to give their feedback. Based on these feedback sessions, the tool was modified with adding new smell definitions. This process was iterated with model calibration until there were no longer suggestions by professionals.

- *Online:* This type of evaluation is done when the system is used by real users that perform real tasks. It is most trustworthy to compare different ranking algorithms online, learn user's context or evaluate user interface, yet not very easy to apply with real users in a software organization. Online evaluation are by nature feasible after the recommendation system is deployed. For example, a code completion recommendation system can incorporate two different algorithms (providing two different recommendations) and randomly switch between them while software developer is using the IDE, and at the same the system can log user interaction (i.e., to learn which algorithm's recommendation is useful) for calibration.
  One example for online experiments is reported in [54], which proposed a recommendation system called *CodeBroker* to locate and propose reusable components by using developers' partially written programs. The system build user models, and implicitly updates them when it observes that software developers reuse a component during programming. Components that have been reused more than X times by a developer are considered as well known and they are not recommended to the developer anymore.

1.2.4 Phase V: Deployment of the recommendation system

Deployment is hard and it requires trust to the recommendation system. Therefore, we have seen very few examples in the literature showing evaluations based on deployed models and real usage after conducting a field study. Main points in the process of deployment can be summarized as follows:

- *Integration with existing systems:* Depending on the design of a recommendation system, it can operate either independently as a standalone application (e.g. [5]) or as a plug-in (e.g. [21, 14]) closely attached to development environments or other systems that are actively used in software organization. In both cases, the system should be fed with data required for operation. Therefore, as the first step before deployment, all systems which a recommendation system communicates with should be clarified by discussing with development teams. Often, this communication has been handled during a pilot study, but scaling up to large systems may need a new design.
  For example, in [21], a tool, *Mylar*, is presented that aims to improve the productivity by monitoring programmer's activity through filtered and ranked information presented in a development environment. Authors validated the performance of Mylar in a longitudinal study and the deployed version has currently been in use by thousands of programmers. During integration, authors built a

bridge architecture to handle the communication and interaction between existing systems, historical storage and Mylar. They also considered the performance of communication between systems by generating mechanisms for storing and collapsing interaction logs.

- *Post-usage calibration:* Even though a recommendation system is calibrated multiple times during offline experiments and user studies, new requests may arise after deployment. In the case of *Mylar* [21], authors realized that daily usage revealed uncovered tasks and misconceptions about how developers work on related tasks. So, they calibrated the tool by adding clusters to cover related tasks. In other systems, such as *DebugAdvisor* deployed in Microsoft [2], tool can be calibrated based on qualitative feedback gathered from users through surveys as well as quantitative data collected from usage logs. In DebugAdvisor, authors let users flag recommendations of the tool as useful or not through a user interface, and the system automatically adjusts itself in future recommendations.
  As mentioned earlier, online experiments are intended to calibrate a recommendation system in real time, which can only be done after the system is deployed. Thus, data stored in logs as users select, rank or filter some of the recommendations (e.g. CodeBroker in [54]), can be used to calibrate the deployed system simultaneously, compared to post-usage feedback in the form of surveys.

2 Challenges in Conducting Field Studies

In Section 1, we describe each of five steps required to build a recommendation system in a real setting. During this step-by-step process, researchers can encounter several challenges which may slow down or interrupt the activities. These challenges can be classified into three: Organizational challenges, data collection & quality challenges, and design challenges.

Organizational challenges reveal themselves mainly in the form of resistance of software teams to actions required to take (conducting interviews, surveys with the team, or asking manual work from the team) for building recommendation systems in a software organization, or in the form of cultural issues, effect of size, or software development methodology used that would affect the process of building such systems.

Data challenges are potential problems that can be encountered while accessing data repositories, or related to the quality or accuracy of data that are extracted from these information sources. Finally, design challenges are related to issues, which should be considered before building a recommendation system, affecting the performance, usability and reliability of its output.

In this section, we will give examples for these challenges at each step of this whole process.

2.1 Organizational challenges

2.1.1 Phase I

Initial meetings with senior management are very essential to define business needs and negotiate about the recommendation system that will be proposed. However it is not sufficient to convince only senior management in software organizations. The success of a field study highly depends on willingness of participants from all levels (e.g. project managers, designers, analysts, developers and testers) who take part during construction, calibration and usage of a recommendation system. Technical staff is more likely to know about software artefacts that are actively used during development (e.g. tools, usage logs, documentation, configuration management systems), which are also required to build a recommendation system. Moreover, recommendation systems often target development team as real users, and hence, their initial motivation and knowledge about the context, process and possible risks would ease the construction and deployment of these systems. A large kick-off meeting consisting of software engineers (i.e. designers, analysts, developers and testers), junior and senior managers as well as researchers is necessary to overcome these challenges at initiating a field study. It is crucial that the recommender system aims to make life easier for software engineers (e.g. analysis, designers, developers, testers, etc.). Otherwise, it is impossible to integrate the usage of a recommender system into a company's development life cycle, even if senior management approves the recommender system during initial meetings. It is very likely that resistance of software engineers towards the usage of the recommender system will affect the decisions of senior management in the long run. Therefore, while building recommender systems with a "bottom-up (from development team to managers) approach should be preferred than "top-down (from managers to team) approach.

A recommender system will be useful,only if it provides solutions for company's existing and significant software engineering problems. Therefore, identification of these problems is crucial for the success of the field study. In order to find out the needs of the company during Phase I, researchers can prefer to use one or more of the following techniques: Brainstorming sessions, focus groups, interviews or questionnaires. Below, we summarize the potential challenges regarding these techniques:

Brainstorming and Focus Groups: These techniques are the most suitable, when researchers are new to a domain and seeking for further explanations. However, it is often hard to schedule a brainstorming session or focus group due to busy schedules of software engineers. Even after scheduling issues are resolved, there are other challenges, which need to be tackled, during brainstorming and focus group sessions. Unless the moderator is well trained, brainstorming and focus groups can become too unfocussed. Moreover, people are not perfect recorders of events around them. Due to their biases and perceptions, people preferentially remember events that are meaningful to them. Thus, they assess the problems, which they experience more significant compared to the other software engineering prob-

lems in the company. For instance, according to a test lead, allocation of testing resources may be a more significant issue compared to others. On the other hand, from senior management's point of view, the most significant problem may be the cost estimation of a software project. Therefore, during initial meetings, the main problem that will be addressed in a recommendation system should be clearly specified. Triangulation techniques can be used to mitigate the problems arising due to the biased data, which is obtained directly from software engineers. For this purpose, in addition to brainstorming sessions and focus groups, one can employ methods such as questionnaires, interviews, observation/shadowing, documentation analysis and investigation of data stored in work databases, version and issue management systems.

Another issue arises due to sample of participants in these meetings, i.e., convenience sampling. Software engineers and managers who are voluntarily involved to these meetings and sessions may cause a self-selection bias [24, 40], because they may have different characteristics from the whole population in a software organization. Therefore, problems indicated by these participants may not reflect company's actual software engineering problems. Conflicts arising due to convenience sampling can be resolved through the analysis of company's annual reports and by consulting senior managers in the organization.

Interviews and Questionnaires: Interviews and questionnaires are often conducted in the same series of studies, however interviews provides additional information to the answers from the questionnaires [24]. Interviews and questionnaires rely on self-assessment of respondents about their habits and attitudes. Therefore, potential challenges in brainstorming sessions and focus groups can also be encountered during interviews and questionnaires (i.e. convenience sampling and availability bias). For instance, in one of the questionnaires conducted by Lethbridge et al. [24], software engineers reported that reading documentation was a time-consuming aspect of their job. However, when the researchers observed software engineers when they are engaged in their work for 40 hours, they hardly saw anyone doing so. Thus, the information was biased due to the fact that engineers tend to remember events that are meaningful to them.

2.1.2 Phase II

Data collection techniques, which are employed to build recommender systems, generally involve analysis of documents, source code and history logs, that are obtained from work databases, version and issue management systems. Yet, there are situations when recommendation systems may also require human intervention, i.e. direct or indirect intervention from real users, especially for evaluating the systems or further calibration. In such situations, user opinions can be collected through questionnaires and interviews. However, interviews are costly and time consuming. Prior to interviews, appointments needs to be scheduled, and usually the researcher needs to spend a lot of time in the field to collect responses of practitioners. Software engineers, on the other hand, will exhibit resistance to such interviews due to

their heavy workloads and tight schedules while rushing for the next release of the software product.

Another issue arising from organizational challenges is the "missing data problem". Data collected to feed the recommendation system might be partially or completely missing in the field, or there might be no defined process inside development life cycle that enforces software engineers for manual entry of comments and required information to company's work databases, such as version and issue management systems. For instance, recommender systems which predict software defect proneness might suffer from lack of information regarding defective files in organizations where changes (commits) in the version management system are not matched with bugs. In such cases, researchers may propose a process change in software development life cycle in order to handle missing data. However, it is very likely that development team would not volunteer to increase their daily workload.

In large software engineering organizations, the work performed by software engineers is often managed carefully through problem reporting, change request and configuration management systems. The rest of the information (e.g. descriptions and comments) needs to be entered to systems manually. When there is little control over the quantity and quality of manually entered information, it is very likely to encounter descriptive fields, which were not filled in. Moreover, some fields might be filled in different ways by different developers. Comments entered by developers might contain cryptic abbreviations, which would be expected to form a consistent picture only in the minds of the software engineers who originally wrote them. In such cases, one solution would be to consult the comment owners. However, there are also some situations when a record is old or the software engineer who worked on it is no longer available.

2.1.3 Phase III & IV

A successful recommendation system should align with the goals of the organization in order to be successful. During Phase III, developers should work with practitioners to ensure this alignment. In this section, we discuss the challenges related to the development methodology, organization size and culture of the organization that may affect Phases III & IV of a recommendation system.

Software development methodologies: In order to align with the organizational goals, a recommendation system should target one or more parts of a development methodology and provide a cost effective alternative by automating or reducing the workload related to those parts. The employed software development methodology provides clues for solving some of the concerns of the organization. It should be noted that particular implementation of a methodology in an organization may differ significantly from its textbook definition. Therefore, it is beneficial to observe the actual setting or at least the most relevant phase related to the recommendation system.

Release cycles and development phases change dramatically between iterative/ agile and waterfall-like methodologies [45]. In traditional methodologies, like wa-

terfall, design, development and testing are clearly separated at least in theory. In this case, providing detailed reports to guide and increase the efficiency of the next phase may be beneficial. However, in agile development, software phases tend to overlap with each other, and instant recommendations during the iterations may be a better approach compared to recommendations for each phase.

Rascal, a recommendation system developed by McCarey et al. to propose reusable code snippets from the source code repository is an example of such approaches [29]. Authors initially discussed various shortcomings of the agile methodology, such as lack of support materials and documentation in agile projects. Afterwards, they showed how their recommendation system addresses the problem of lack of support materials by proposing reusable snippets in the source code to developers with no support documents.

Organization Size: Size (the number of people involved) is an important characteristic of groups, organizations, and communities in which social behavior occurs. Size is also an indicator about the complexity of an organization [13].

Organization size may affect the organizational needs in several ways. The complexity that emerges with size hinders the capacity of the organization to change fast. Every small change may be politically challenged by cliques that attempt to keep the status quo [12]. These political challenges may complicate getting organizational data to calibrate the recommendation system and setting up (implementation and integration) in software organizations. In such cases trust metrics may be defined for a recommendation system to estimate its real value for an organization [28].

Organizational Culture: Culture is a broad term with different definitions proposed by different sociologists. For instance, William B. Gudykunst defines culture as the systems of knowledge shared by a relatively large group of people [17]. In order to build a successful recommendation system, one should understand the organization's systems of knowledge, i.e., the organization culture. Organization culture may affect the knowledge needs and user interface expectations. For this reason, Chen et al. proposes a survey based approach to understand the key organizational characteristics before building a recommender [7].

Sociologist Geert Hofstede defines five dimensions in the organizational culture [20]. Three of the proposed dimensions affect the capacity of change in an organization:

- *Term Orientation:* Term orientation is the relative importance of future for an organization [20]. Long term oriented organizations attach more importance to the future and foster pragmatic values oriented towards rewards, including persistence, saving and capacity for adaptation. In short term oriented organizations, values promoted are related to the past and the present, such as steadiness and respect for tradition. Long term oriented organizations introduces new ideas to the organization such as a new recommender system relatively easier than short term oriented organizations.
- *Uncertainty avoidance index:* Uncertainty avoidance index defines the capacity and willingness of organizations to deal with an uncertain environment [20]. Uncertainty avoidance index may affect the expectation of a company from a rec-

ommender system. Organizations with high uncertainty avoidance index try to minimize the occurrence of unknown and unusual circumstances and to proceed with careful changes step by step by planning and by implementing rules, laws and regulations. In contrast, low uncertainty avoidance organizations accept and feel comfortable in unstructured situations or changeable environments and try to have as few rules as possible.

- *Power distance index (PDI):* Power distance is the extent to which the less powerful members of organizations accept and expect that power is distributed unequally [20]. Organizations with high PDI depend on the decisions of a few influential people. In these organizations, number of people that must be convinced about the benefit of a recommender is low but if the influential backers of the project leaves there is a risk of losing support. On the other hand, organizations with low PDI tend to make the decisions collaboratively. In these organizations, convincing the entire organization about benefits of a recommendation system is harder, but the risk of employee turnover affecting the support is low.

2.1.4 Phase V

The main goal of recommendation systems in software engineering is to help developers through decision making, by intelligently narrowing down all possible alternatives to solve a specific problem, and providing simple, easy-to-use and actionable recommendations. While doing that, recommendation systems should communicate with existing systems to collect data, analyse patterns, run its prediction engine, store results and report through a user-friendly interface. Unfortunately, due to complexity of the engine and operations, these systems become too complex and cumbersome.

Besides, these systems often require calibration, in the form of updating the input (data) or adjusting the granularity of the output, at several times during their usage. However, in many software organizations, there is almost no qualified personnel as data analysts, who can understand the underlying mechanism of recommendation systems, evaluate outputs and decide when to update input data, how to improve the performance, and take actions accordingly. Combination of lack of experts in software organizations and complex and heavy systems affects not only the deployment phase, but also post-deployment/usage phase. Possible solutions to these challenges can be to design systems with simple and actionable inputs as well as self-adjustable (automatic calibration) based on new data or user feedback.

In [33], authors argued that even though precision of recommendation systems is high and reliable, users in a software organization may not trust on the recommendations. Thus, they may not feel like recommendations are useful and help them solve some of their challenges in daily activities. To increase trust, and hence, usage of these systems, software professionals offered some solutions in [33], such as simulating social interactions between peers in a recommendation system, or giving recommendations based on what peers in development teams do or other experts do.

2.2 Data challenges

2.2.1 Phase II

"Security and privacy issues" are very likely to arise, since building recommendation systems requires accessing active systems of software organizations, such as source code, documentation, version management systems. Organizations may not be willing to share such data with researchers, due to security and privacy issues. Therefore, during initial meetings and discussions, solutions should be found to collect data without violating company's privacy and security policies. For instance, one or two people form company's technical staff can be trained to extract required data from available resources and share anonymously with researchers.

In addition to being related to organizational challenges, "missing data problem" can also be categorized as a data challenge. Even though organizational challenges are resolved (i.e. the development team agrees upon a change in the development process), it will take time to collect enough data to build a recommender system.

For example, in [47], data collection step was problematic at the beginning, since changes in version control systems were not properly matched with bugs. Researchers called for an emergency meeting with senior managers and development team to explain the problem and its effects on building a recommender system to predict pre-release software defects. After this meeting, a process change has been adopted, initially at pilot projects, such that developers have to provide issue ID and additional comments about the reason of change, when they make a commit to the version control system. However, since data collection would take significant amount of time, in order to minimize time loss, researchers had to propose an alternative solution. This alternative solution was using cross-company (CC) data [49], in order to build a prototype recommender system. In other words, a two-phase approach was employed: 1) Researchers used imported CC data, which was filtered via Nearest Neighbourhood (NN) algorithm, to build the recommendation system. 2) The organization started a data collection program. Phase two commenced when there was enough local data (i.e., Within Company (WC)) to build the final system. During phase two, the organization would switch to new defect predictors learned from the WC data.

The problem of missing data is also encountered in datasets, which are used to build recommendation systems estimating a project's cost/effort [34], [51]. In empirical software engineering literature, some imputation methods have been employed to handle missing data problem in cost estimation models. Imputing means to fill missing values by considering the underlying missing value mechanism [26]. There are various techniques, which are used in to handle missing data in software cost estimation models [34], [51], [50], [6], [46]. In most case data is missing due to high data collection costs. This usually is the case in software cost estimation, since the cost of gathering and reporting data from software projects is non-negligible [11]. Expectation Maximization methods such as EMMI and EMSI are very powerful techniques [51], [26]. According to the results obtained in the benchmarking analy-

sis by Twala et al., EMMI gave the highest accuracy results with an excess error of 0.04 [51].

2.3 Design challenges

2.3.1 Phase III & IV

There are many potential design challenges during the building phase of recommendation systems. In this section we provide a partial list of common challenges based on our experience:

- **Scaling challenges:**
  Large organizations tend to have complex and large software repositories and hence, dedicated specialists for configuration management. Large software repositories provide rich input data to recommendation systems, yet cause different challenges during feeding the system with historical data. A system tested on a 100 kLOC repository developed by 5 people may not work for a 10 mLOC repository with multiple branches and 10+ years of development history developed by 1000 people.
  Scalability is an important challenge for recommendation systems just like any other software. Some algorithms scale easily while others fail dramatically when scaled even on most advanced hardware. On the other hand, data storage needs may get very high in a real usage scenario or number of concurrent users may freeze the system. If the recommender runs on a client machine, scalability problems may force developer to terminate the application rather than waiting for a response. It is easier to address possible scalability issues during design of recommendation systems in software engineering. Client-server workload distribution, multi-machine scaling availability, data storage estimation are some of these issues that must be considered in order to avoid scalability problems [27].
- **Data privacy challenges:** Privacy is a common concern for everyone in the digital age. There should be a frank explanation of what the recommendation system can and will do with the extracted data in order to avoid the privacy concerns. If a clear privacy statement is not in effect, people may think the software as a modern big brother microphone even for the most innocent recommendation systems.Therefore, every recommender system should have a clear and transparent data privacy policy [36].
- **Usability:** Usability is a common issue for software systems. A recommendation system should be easy to configure and should provide actionable outputs. Visualization, live notifications and a user-friendly interface may dramatically affect the usage of the system. In this regard, observing work patterns of real users of a recommendation system, and early mock-up designs help researchers to test and improve the usability of the system [23].
- **Is the solution too local?** Specific artificial scenarios may overestimate the model performance during beta testing. User trust to a recommendation system

may significantly degrade, if the system fails in very common, yet simple scenarios in reality [38].

2.3.2 Phase V

Design of recommendation systems in terms of input, output and the engine is explained in Section 1.2.3. Nevertheless, a redesign approach may be needed before deployment. During this process, researchers often come across challenges, such as scalability, privacy, robustness, adaptivity [38]. Privacy in terms of data and user profiles and scalability have already been discussed in sections 2.2.1 and 2.1.3. Robustness is necessary to provide stability in the presence of fake information provided by developers. Sometimes, developers may try to fake such systems to see how they behave in the presence of noisy or wrong data. Another situation can be to provide stability under conditions when there are lots of user requests. Before deployment, tests are done with a small set of practitioners, and hence, the system may not operate properly when its client runs on hundreds of developer machines.

Adaptivity is another issue related with rapid changes or shifts in the input data provided to recommendation systems. In general, recommendation systems should adapt itself to unusual events, like a system recommending news to readers may shift its focus for a short period of time due to a disaster [38]. In software engineering, a large scale project with many connected components may produce a lot more development logs than usual, and hence, it may require different recommendations on topics, such as failure-proneness or reusable components. In such cases, a recommendations system should adapt itself to such large amounts of data through filtering or adjustments in existing algorithms. In some cases, additional offline experiments can also be conducted to compare different algorithms.

3 Methodological Issues

In empirical software engineering research, research design should be consistent and follow certain protocols depending on the research methodology. In this section, we will mention potential threats that must be addressed in a field study in order to increase the validity and reliability of results. It must be noted that there are many aspects that need to be considered in an empirical research depending on the methodology (e.g. survey in an organization, field experiments for comparing different approaches, multiple case studies), and hence, we ask the reader to follow empirical software engineering literature, such as [53, 52, 22].

During building a system, tool or experimenting a new technique in the field, it may not be necessary to derive general conclusions. However, depending on the experimentation type (e.g. online experiments, offline experiments, user studies), possible threats to the validity of results should be considered as early as possible, generally at the planning phase. For example, when an initial prototype of a system

is going to be built using offline experiments, internal, construct and conclusion validity should be considered ([53], p.102). Or when a qualitative analysis as a case study is conducted to make early analysis on the prototype, case study protocols, hypothesis analysis and other factors should be considered [55]. If researchers aim to achieve more general conclusions based on multiple case studies or the field usage, external validity issues should be considered [53]. Moreover, research design with cause and effect constructs should be selected by considering replication of the same design on different setting ([40], p.365).

Field studies require involvement of human subjects, in addition to collection of information that can help identifying individuals through examination of software artefacts (e.g. source code and documents). Therefore, ethical issues should be taken into consideration. Researchers, who do not follow mandated ethical guidelines, risk losing their cooperation or honesty [41], as well as losing access to funding and other resources [42]. In empirical software engineering literature, there are some guidelines, which can help researchers to deal with ethical issues, while conducting field studies, [43], [44], [52].

In a field study, there is one major issue affecting the whole process from planning to design of experiments or systems/tools, or from evaluation of results to deployment: Researchers. It is very critical to have domain knowledge, i.e., characteristics and problems of the domain, experience about organizational dynamics, or have the ability to observe and interpret activities through qualitative and quantitative data in a field study. Researchers who aim to implement recommendation systems in real settings must combine their knowledge about empirical research methodologies with their previous knowledge about dynamics of software development, and interpret what they observe to provide useful and practical solutions for practitioners. In addition, researchers must be careful about the impact of their own bias when preparing questionnaires, conducting surveys with practitioners and interpreting analysis results. For example, in [40] (p. 77), researcher bias is briefly defined and suggestions are given to avoid this throughout questionnaire construction.

4 What is Next?

In this chapter we have discussed field studies in the context of recommendation systems in software engineering. Our aim is to guide researchers step by step how to conduct a field study in building recommendation systems. At the beginning of this chapter, we highlighted the importance of having a research thought process and its consistency in order to put field studies into perspective. Then, we highlighted potential challenges throughout this process and suggested solutions to overcome some of these challenges in software engineering. Our research have been on building recommendation systems with industry partners for years. We have conducted many field studies, and deployed recommendation systems in the industry. Some of these models/ systems were successful, and some were not. The aftermath of our

experiences over the years show that conducting field studies is a rewarding experience both for researchers and practitioners. Software engineering, specifically the area of recommendation systems, is a rare topic where researchers and practitioners can work together to solve problems in the domain. We suggest that recommendation systems should be built for the men on the field (analyst, developer, architect, tester) as part of their routine tasks as tool support/ plug-in [31]. We envision that such a plug-in should give reasons regarding causalities that are mapped to business objectives and hence, the output of a recommendation system should be easy to interpret and actionable. Field studies is the only way researchers can understand the domain and propose technical solutions for real needs of practitioners. We have seen that it is easy to overcome technical challenges, however, social, organizational and cognitive challenges are the ones that make or break the adoption and usage of recommendation systems difficult [31]. That is why we need to conduct more field studies to explore what is out there, and how we can simplify a problem and propose simple solutions. Field studies will help us understand some of the areas that were overlooked such as modelling people aspects of software development in building recommendation systems, or combining human social interactions with code dependency structure to connect the dots.

Lack of generalizations in field studies is a long debated topic in empirical software engineering. Isolated case studies only show benefits of a recommendation system in a particular context. However, as researchers we can use the synergies of other techniques such as machine learning/ data mining to find common patterns, contexts and make generalizations from local models. Such a combination of software engineering and machine learning can help decision makers to access enormous amount of data for analysis and to remove errors in their thinking process. Researchers by building recommendation systems as tools would enable practitioners to query a huge database of different contexts and organizations for making recommendations [5, 48].Recommendation systems should not be built as offline number crunching experiments. In order to build a theory, we need to understand the underlying concepts, and combine them with available data and models [48]. In this sense, field studies are the only way to discover concepts, assumptions and limitations.

References

1. Anvik, J., Murphy, G.C.: Reducing the effort of bug report triage: Recommenders for development-oriented decisions. ACM Trans. Softw. Eng. Methodol. **20**(3), 10:1–10:35 (2011). DOI 10.1145/2000791.2000794. URL http://doi.acm.org/10.1145/2000791.2000794
2. Ashok, B., Joy, J., Liang, H., Rajamani, S.K., Srinivasa, G., Vangala, V.: Debugadvisor: a recommender system for debugging. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE '09, pp. 373–382. ACM, New York, NY, USA (2009). DOI 10.1145/1595696.1595766. URL http://doi.acm.org/10.1145/1595696.1595766
3. Bakir, A., Kocaguneli, E., Tosun, A., Bener, A., Turhan, B.: Xiruxe: An intelligent fault tracking tool. International Conference on Artificial Intelligence and Pattern Recognition (AIPR)

pp. 293–300 (2009)
4. Bruch, M., Monperrus, M., Mezini, M.: Learning from examples to improve code completion systems. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE '09, pp. 213–222. ACM, New York, NY, USA (2009). DOI 10.1145/1595696.1595728. URL http://doi.acm.org/10.1145/1595696.1595728
5. Caglayan, B., Misirli, A.T., Calikli, G., Bener, A., Aytac, T., Turhan, B.: Dione: an integrated measurement and defect prediction solution. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 20:1–20:2. ACM, New York, NY, USA (2012). DOI 10.1145/2393596.2393619. URL http://doi.acm.org/10.1145/2393596.2393619
6. Cartwright, M.H., Shepperd, M.J., Song, Q.: Dealing with missing software project data. In: Proceedings of the 9th International Symposium on Software Metrics, METRICS '03, pp. 154–. IEEE Computer Society, Washington, DC, USA (2003). URL http://dl.acm.org/citation.cfm?id=942804.943773
7. Chen, L., Pu, P.: A cross-cultural user evaluation of product recommender interfaces. In: Proceedings of the 2008 ACM conference on Recommender systems, RecSys '08, pp. 75–82. ACM, New York, NY, USA (2008). DOI 10.1145/1454008.1454022. URL http://doi.acm.org/10.1145/1454008.1454022
8. Copeland, T.: Detecting duplicate code with pmd's cpd (2003). URL http://www.onjava.com/pub/a/onjava/2003/03/12/pmd_cpd.html
9. Creswell, J.W.: Research Design: Qualitative, Quantitative, and Mixed Methods Approaches, 3rd edn. Sage, Los Angeles (2009)
10. Crotty, M.J.: The foundations of social research: Meaning and perspective in the research process. Sage, London (1998)
11. DeMarco, T.: Controlling Software Projects: Management, Measurement, and Estimates. Prentice-Hall; 1 edition (1982)
12. DeMarco, T., Lister, T.: Peopleware: Productive Projects and Teams, 2nd edn. Dorset House Publishing Company (1999)
13. Dewar, R., Hage, J.: Size, technology, complexity, and structural differentiation : Toward a theoretical synthesis. Administrative Science Quarterly **23**(1), 111–136 (1978). URL http://www.jstor.org/stable/10.2307/2392436
14. Duala-Ekoko, E., Robillard, M.: Using structure-based recommendations to facilitate discoverability in apis. In: ECOOP 2011–Object-Oriented Programming, pp. 79–104 (2011)
15. Duvall, P.: Automation for the people: Improving code with eclipse plugins. Tech. rep., Stelligent Incorporated (2007). URL http://www.ibm.com/developerworks/java/library/j-ap01117/index.html
16. Guba, E.G.: The paradigm dialog. Sage, Beverly Hills, CA (1990)
17. Gudykunst, W., Ting-Toomey, S., Chua, E.: Culture and interpersonal communication. Sage series in interpersonal communication. Sage Publications (1988). URL http://books.google.com.tr/books?id=tiBHAAAAMAAJ
18. Guo, Y., Seaman, C., Zazworka, N., Shull, F.: Domain-specific tailoring of code smells: an empirical study. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10, pp. 167–170. ACM, New York, NY, USA (2010). DOI 10.1145/1810295.1810321. URL http://doi.acm.org/10.1145/1810295.1810321
19. Han, S., Dang, Y., Ge, S., Zhang, D., Xie, T.: Performance debugging in the large via mining millions of stack traces. In: 34th International Conference on Software Engineering (ICSE), pp. 145–155 (2012)
20. Hofstedte, G.: Cultures and Organizations: Software of the Mind. McGraw-Hill Professional; 3 edition (2010)
21. Kersten, M., Murphy, G.C.: Using task context to improve programmer productivity. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, SIGSOFT '06/FSE-14, pp. 1–11. ACM, New York, NY, USA (2006). DOI 10.1145/1181775.1181777. URL http://doi.acm.org/10.1145/1181775.1181777

22. Klein, H.K., Myers, M.D.: A set of principles for conducting and evaluating interpretive field studies in information systems. MIS Q. **23**(1), 67–93 (1999). DOI 10.2307/249410. URL http://dx.doi.org/10.2307/249410
23. Krug, S.: Don't Make Me Think: A Common Sense Approach to the Web (2nd Edition). New Riders Publishing, Thousand Oaks, CA, USA (2005)
24. Lethbridge, T.C., Sim, S.E., Singer, J.: Studying software engineers: Data collection techniques for software field studies. Empirical Software Engineering **10**, 311–341 (2005)
25. Lincoln, Y.G., Guba, E.G.: Naturalistic Inquiry. Sage, Beverly Hills, CA (1985)
26. Little, R.J., Rubin Donald, B.: Statistical Analysis with Missing Data. Wiley; 2 edition (2002)
27. Liu, H.H.: Software Performance and Scalability: A Quantitative Approach (Quantitative Software Engineering Series). Wiley (2009)
28. Massa, P., Bhattacharjee, B.: Using trust in recommender systems: An experimental analysis. In: C. Jensen, S. Poslad, T. Dimitrakos (eds.) Trust Management, *Lecture Notes in Computer Science*, vol. 2995, pp. 221–235. Springer Berlin Heidelberg (2004)
29. Mccarey, F., Cinneide, M.O., Kushmerick, N.: Rascal: A recommender agent for agile reuse. Artificial Intelligence Review (2005)
30. Mertens, D.M.: Research methods in education and psychology: Integrating diversity with quantitative & qualitative approaches. Thousand Oaks: Sage (1998)
31. Misirli, A.T., Caglayan, B., Bener, A., Turhan, B.: A retrospective study of software analytics projects: In-depth interviews with practitioners. IEEE Software (2013)
32. Mockus, A., Herbsleb, J.D.: Expertise browser: a quantitative approach to identifying expertise. In: Proceedings of the 24th International Conference on Software Engineering, ICSE '02, pp. 503–512. ACM, New York, NY, USA (2002). DOI 10.1145/581339.581401. URL http://doi.acm.org/10.1145/581339.581401
33. Murphy, G.C., Murphy-Hill, E.: What is trust in a recommender for software development? In: Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, RSSE '10, pp. 57–58. ACM, New York, NY, USA (2010). DOI 10.1145/1808920.1808934. URL http://doi.acm.org/10.1145/1808920.1808934
34. Myrtveit, I., Stensrud, E., Olsson, U.H.: Analyzing data sets with missing data: An empirical evaluation of imputation methods and likelihood-based methods. IEEE Transactions on Software Engineering **27**(11), 999–1013 (2001). URL http://ieeexplore.ieee.org/stamp/965340
35. Neuman, W.L.: Social research methods: Qualitative and quantitative approaches, 4 edn. Allyn&Bacon, Boston (2000)
36. Ramakrishnan, N., Keller, B.J., Mirza, B.J., Grama, A.Y., Karypis, G.: Privacy risks in recommender systems. IEEE Internet Computing **5**(6), 54–62 (2001). DOI 10.1109/4236.968832. URL http://dx.doi.org/10.1109/4236.968832
37. Robillard, M.P., Walker, R., Zimmermann, T.: Recommendation systems for software engineering. IEEE Software pp. 80–86 (2010)
38. Shani, G., Gunawardana, A.: Evaluating Recommendation Systems, pp. 257–297 (2011)
39. Shull, F.: Research 2.0. IEEE Software (2012)
40. Shull, F., Singer, J., I.K.Sjoberg, D. (eds.): Guide to Advanced Empirical Software Engineering. Springer-Verlag (2008)
41. Sieber, J.E.: Planning Ethically Responsible Research: A Guide for Students and Internal Review Boards, 1st edn. SAGE Publications (1992)
42. Sieber, J.E.: Protecting research subjects, employees and researchers: Implications for software engineering. Empirical Software Engineering **6**(4), 329–341 (2001). DOI 10.1023/A:1011978700481. URL http://dx.doi.org/10.1023/A:1011978700481
43. Singer, J., Vinson, N.G.: Why and how research ethics matters to you. yes, you! Empirical Software Engineering **6**(4), 287–290 (2001). DOI 10.1023/A:1011998412776. URL http://dx.doi.org/10.1023/A:1011998412776
44. Singer, J., Vinson, N.G.: Ethical issues in empirical studies of software engineering. IEEE Transactions on Software Engineering **28**(12), 1171–1180 (2002). URL http://dx.doi.org/10.1109/TSE.2002.1158289
45. Sommerville, I.: Software engineering (5th ed.). Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1995)

46. Strike, K., El-Emam, K., Madhavji, N.: Software cost estimation with incomplete data. IEEE Transactions on Software Engineering **27**(10), 890–908 (2001). URL http://doi.ieeecomputersociety.org/10.1109/32.935855
47. Tosun, A., Bener, A.B., Turhan, B., Menzies, T.: Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry. Information and Software Technology **52**(11), 1242–1257 (2010)
48. Turhan, B., Bener, A.: On combining the scattered knowledge: Putting the bricks together. In: 2nd International NSF sponsored Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE: ICSE Workshop) (2013)
49. Turhan, B., Bener, A., Menzies, T.: Nearest neighbor sampling for cross company defect predictors. In: Proceedings of the 1st International Workshop on Defects in Large Software Systems, DEFECTS '08, pp. 26–26. ACM, New York, NY, USA (2008). DOI 10.1145/1390817.1390824. URL http://dl.acm.org/citation.cfm?doid=1390817.1390824
50. Twala, B.: An empirical comparison of techniques for handling incomplete data using decision trees. Journal of Applied Artificial Intelligence **23**(33), 373–405 (2009)
51. Twala, B., Cartwright, M., Shepperd, M.: Ensemble of missing data techniques to improve software prediction accuracy. In: Proceedings of the 28 ACM/IEEE International Conference on Software Engineering, ICSE '06, pp. 909–912. ACM, New York, NY, USA (2006). DOI 10.1145/1134285.1134449. URL http://doi.acm.org/10.1145/1134285.1134449
52. Vinson, N.G., Singer, J.: Guide to Advanced Empirical Software Engineering, chap. A Practical Guide to Ethical Research Involving Humans, pp. 229–256. Springer (2008)
53. Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., Wesslen, A.: Experimentation in Software Engineering. Spring (2012)
54. Ye, Y., Fischer, G.: Supporting reuse by delivering task-relevant and personalized information. In: Proceedings of the 24th International Conference on Software Engineering, ICSE '02, pp. 513–523. ACM, New York, NY, USA (2002). DOI 10.1145/581339.581402. URL http://doi.acm.org/10.1145/581339.581402
55. Yin, R.K.: Case study research : design and methods, 3rd edn. Sage Publications (2003). URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0761925538