Engineering Big Data Solutions

Audris Mockus
Avaya Labs Research
211 Mt Airy Rd, Basking Ridge, NJ, USA
audris@avaya.com

ABSTRACT

Structured and unstructured data in operational support tools have long been prevalent in software engineering. Similar data is now becoming widely available in other domains. Software systems that utilize such operational data (OD) to help with software design and maintenance activities are increasingly being built despite the difficulties of drawing valid conclusions from disparate and low-quality data and the continuing evolution of operational support tools. This paper proposes systematizing approaches to the engineering of OD-based systems. To prioritize and structure research areas we consider historic developments, such as big data hype; synthesize defining features of OD, such as confounded measures and unobserved context; and discuss emerging new applications, such as diverse and large OD collections and extremely short development intervals. To sustain the credibility of OD-based systems more research will be needed to investigate effective existing approaches and to synthesize novel, OD-specific engineering principles.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Process Metrics, Software science; D.2.4 [**Software/Program Verification**]: Statistical methods; H.2.5,8 [**Information Systems**]: Heterogeneous Databases, Database ApplicationsData translation, Data mining; I.7.1 [**Document and Text Editing**]: Version control

General Terms

Measurement, Management, Experimentation, Economics, Human Factors

Keywords

Analytics, Operational Data, Data Quality, Game Theory, Statistics, Data Engineering, Data Science

1. INTRODUCTION

Many human activities are increasingly transacted partially or entirely within the digital domain, generating voluminous traces that wait to be explored, understood, and used. Software engineering has been an early beneficiary of this phenomenon. Operational support tools, such as version control and issue tracking, were built to support common tasks and over time have generated transactions and logs that have been successfully used to understand and improve software development. We discuss research directions needed to sustain and expand this success.

This paper concerns Operational Data (OD): data left as traces from multiple operational support tools and then integrated with more traditional data. Section 2 discusses OD and other flavors of big data. Operational Data Solutions (ODS) are (software) applications that utilize OD to provide insights or tools. We discuss domains where ODS would be particularly helpful and argue for the need to systematize ways to engineer ODS to ensure the integrity of the results produced by these systems.

Three basic trends affecting ODS are likely to continue: evolution of Version Control Systems (VCSs) and Issue Tracking Systems (ITSs) and associated practices; increasing use of other operational support tools and associated OD; and expansion to application domains that are not (yet) considered to be software engineering.

It is reasonable to expect that generating and using digital traces will be an integral part of software in many domains, thus it is imperative that practices and tools for engineering ODS be developed and used. How should the engineering principles for ODS be developed? First, existing ODS in software engineering and in other areas can be studied using empirical approaches. Second, engineering paradigms and tools from domains such as databases, statistics, machine learning, social networks, and text analysis will need to be evaluated and adopted for engineering ODS. Finally, synthesizing new approaches from basic principles will also be necessary in cases where the existing or borrowed approaches are inadequate. In particular, effective ways to understand how data are collected, how data should be filtered, augmented, integrated, and modeled, and how to structure ODS to be amenable to verification and maintenance, will need to become an integral part of engineering principles for ODS.

OD originates from traces collected and integrated from a variety of operational support tools. Each event recovered from such traces has a specific context of what may have been on the actor's conscious and subconscious mind, the purpose of that action, the tools and practices used, and the project or the ecosystem involved. Each event, therefore, may have a unique meaning. Some actions, such as verbal communication, may be missing if conducted without tool

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FOSE '14, May 31 - June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2865-4/14/05 ...$15.00.

support. Data filtering or tampering may be done by the actors, operating tools, or data processing and integration at the time of action or later. Determining how to properly segment, filter, augment, and model such data to ensure that they contain a representative sample of relevant activities, is the fundamental challenge of engineering ODS and it will be essential to develop methods to draw valid conclusions from such disparate and low-veracity data.

To have broad effects on software development practices, ODS will need to be not simply a tool for an analyst, but become a part of existing or future operational support tools. Such integrated systems would serve both as data sources for ODS and use ODS results in their regular workflow.

To be used effectively, ODS will require a thought process that can incorporate uncertainty and limitations of the modeling assumptions and would demand a nontrivial understanding of the specific problems to be solved.

Section 2 considers the scope and history of big data in software engineering. Section 3 discusses potential threats to ODS viability and ways to prevent them. Section 4 outlines the defining features of OD to guide the selection of engineering principles. Section 5 considers some of the unique aspects of engineering ODS and Sections 6 and 7 discuss emerging trends that would create specific needs and opportunities for ODS. Finally, Section 8 summarizes the preceding observations into a research roadmap.

2. BIG DATA IN SOFTWARE

It is instructive to consider some of the terms that have been associated with data-driven operation and decision making. Data-driven operation involves actions that may be undertaken manually or automatically in response to a change of certain values of data, e.g., sending notifications to subscribers when the status of an issue is updated. Data-driven decisions are performed by an actor in response to provided information. Actions may be strategic, e.g., to deploy code inspections in response to data indicating that inspections provide substantial benefits. Most actions would be tactical, e.g., to pick a link to follow from the data comprising a list of search results or to select a priority for a new issue report.

Empirical software engineering [3, 2] is an attempt to establish empirical relationships among various software production factors and to use these insights to improve software engineering practice. Business intelligence [58] is a set of theories, methodologies, architectures, and technologies that transform raw data into meaningful and useful information for business purposes. Predictive analytics [60] encompasses a variety of techniques from statistics, modeling, machine learning, and data mining that analyze current and historical facts to make predictions about future, or otherwise unknown, events. The Mining Software Repositories (MSR) field analyzes the rich data available in software repositories to uncover interesting and actionable information about software systems and projects [41]. Big data [57] usually includes data sets with sizes beyond the ability of commonly used software tools to capture, curate, manage, and process the data within a reasonable time. Data science seeks to use all available and relevant data to effectively tell a story that can be easily understood by non-practitioners. It is not unusual to see the term big data used interchangeably with the terms data science, data analytics, business intelligence, machine learning, and statistics.

To avoid the confusion over the exact definition of big data, we use terms OD and ODS defined in the introduction. OD may come in a variety of data formats and types and can be derived from interactions between people and machines, such as web applications, social networks, or issue trackers. OD often lack structure, e.g., are documents, code, or bug descriptions, contain a variety of interrelated entities, are of low veracity (are incomplete or have been tampered with), and are used for unintended purpose. ODS can be thought of as decision support systems utilizing data produced in the course of operations. We also consider ODS primarily within the scope of supporting decision making in software engineering.

The origins of data-driven decision support in software engineering have a long and rich history. The models of software growth [5], reliability [26], productivity [55], and cost [7] appeared about four decades ago. Measures of code complexity [30, 21] and tools to support development (version control) also appeared around the same time [44, 54]. Empirical software engineering with methods of software data collection [3] and applications in industry [18] appeared a decade later. However, collecting data manually was expensive and limited the application of these empirical techniques.

The use of version control [44] and issue tracking tools [32] accumulated a substantial history of software changes. The value of that OD was initially not clear, but a number of publications and applications in industry illustrated its potential in predicting defects [64], visualizing code [14], changes and developers [38], relationships among parts of code [15], or quantifying development practices [33].

Research interests in the area are exemplified by The Metrics Symposium which later became The Empirical Software Engineering and Measurement Conference. These conferences have been running for over two decades. The number and diversity of publications took off with the launch of The International Workshop on Mining Software Repositories [23]. Most recently, a special issue of IEEE Software [22] focuses on ODS in software engineering and IEEE Computer [31] magazine considers some opportunities and problems brought by OD.

The operational support tools will likely evolve in the future to facilitate ODS intelligence capabilities, thus collecting some data explicitly intended for a particular type of analysis. Will there still be a need to analyze OD data for unintended purposes? The following reasons suggest that this need will likely become even greater. First, the operational support tools have to be focused on their primary task of ensuring uninterrupted operation, but the need to collect additional data would substantially decrease their performance and maintainability, making it unlikely that all such needs will be met. Vendors of operational support tools may cater to a wide base of customers and be unwilling to satisfy the needs of a specific project or a domain. Finally, the data needs tend to change over time and expand in scope as new use cases tend to be discovered much faster than the infrastructure could evolve.

The use of OD has benefits and pitfalls. First, the data collection is non-intrusive because operational support tools are already deployed and used in operation. Unfortunately, that does not reduce the need for in-depth understanding of procedures and practices, in particular, of how the operational support tools were used. Effective ways to do such tasks, especially on very large and diverse collections have yet to be developed (see Section 5.1.)

The history of a long-running project or of past projects captured in the tools enables historic comparisons, calibration, and immediate diagnosis in emergency situations (without the need to collect additional data).

The information obtained from the operational support tools is often fine grained, at the trouble ticket or customer installation level. Unfortunately, reliable links between different tools are lacking, as for example, between issue reports and code changes. Different tools often track the same entities at different aggregation levels, for example, sales operations are focused on customers but service operations concern sites and service contracts. Methods to create links between different aggregations and evaluating the impact of dropping unlinked data will have to be developed to utilize more and more diverse sources of data.

OD tends to be complete, because all actions involving operational support tools are recorded. Some actions may, however, be accomplished through other means or may be removed from the system (see Section 4.1). Also, the information about what the action pertains to may be nontrivial to infer and some of the data entries, especially those not essential for the domain of activity, tend to be inconsistently or rarely supplied. To reduce the prevalence of creating misleading inferences from such noisy data, we will need methods that efficiently identify and filter out, or are less affected by, such noise.

The data tends to be uniform over time as the operational support tools, by virtue of being business-critical, are rarely changed to avoid major disruptions. That does not, however, imply that the practices of using these tools don't change over the entire period of interest. Furthermore, many software projects have moved to a new generation of VCSs and ITSs, partly because the newer tools can easily import data from older tools and provide compelling operational benefits. However, the schema and the usage practices often change significantly. The impact of such hybrid data will have to be assessed. For example, a Git [59] repository with some history imported from Subversion [10], and with some of that Subversion history imported from CVS [9].

Even one-person projects contain large volumes of data in the operational support tools making it possible to detect even small effects statistically. This, however, depends on the extractability of the relevant quantities.

The operational support tools are used as a standard part of the project, so the software project is unaffected by experimenter intrusion. However, if such data are used in organizational measurement, it may modify the behavior of individuals or groups. As ODS gain popularity, such effects can extend beyond the measures tracked within an organization. For example, it is likely that on-line behavior may adapt to changes in recruitment practices that take into account on-line activities [51] on, for example, github.com or stackoverflow.com.

3. WHY HYPE, DISILLUSIONMENT?

The interest in big data is now close to the pinnacle of the technology hype curve [16]. Empirical observations indicate that the peak of technology hype is followed by a trough of disillusionment as promises don't deliver.

There are well-founded reasons for the OD hype. First, the economics of computing, communication, and storage allows recording and storing massive amounts of not-well-designed-or-optimized data. For example, median pay for one hour of an engineer's time stayed relatively constant (after adjusting for inflation) since 1960 ($30/hour), but the cost of one million floating point operations on a supercomputer went down 3.9 billion times from 1961 to 2013: the cost for CDC 1604 in 1961 amortized over five years yields 2.7M operations per inflation-adjusted US$, while Tianhe-2 initial cost plus power consumption over five years yields 10.7P operations per US$ in 2013. The reinterpretation of Moore's law for the cost of operation (the original law concerns the number of transistors on integrated circuits) would suggest $2^{\frac{2013-1961}{2}} = 67M$, or a 58 times smaller drop. The drop in price allowed computing to encroach on many areas, such as entertainment, that were not economically feasible when an operation was more than nine orders of magnitude more expensive.

The abundance of inexpensive computing leads to an increasing reliance on software support in many human activities. This software often keeps track of the current state of the system and has the ability to recover from or to report on past states, resulting in massive amounts of digital traces being generated and stored. The collection and analysis of such traces become much simpler to conduct as the software support tools move on-line (to the cloud) and the associated data are collected in a centralized location. It will take some time fully to understand the opportunities afforded by the existing OD, to build the relevant infrastructure and to change the existing culture. New applications and new domains for OD will, consequently, expand in the near future.

At the same time, the hype cycle will likely lead to the trough of disillusionment [16] as experiments and implementations fail to deliver. First, the rapid depreciation of computing may slow in the future as the cost of energy becomes a limiting factor. As a consequence, designing techniques for more efficient storing and processing of data will likely regain importance.

An immediate cause for disillusionment will come from the realization that more data often does not imply more information: the reliance entirely on OD may not work. Instead, a more labor-intensive measurement targeted to collect the most accurate information that is used by itself or to calibrate or interpret the results obtained via OD may be necessary. A better understanding of data quality (see Section 5.4) would need to evolve to counter this threat.

The second immediate cause for disillusionment will be driven by "Big Lies" perpetrated using OD. Most of these lies will be caused by the inexperience of the analysts flocking to this promising area. For example, many of the analytic techniques were designed to work with Gaussian distributions, but in software development such assumptions almost never apply. While some of the basic adaptations, such as considering logarithmically transformed data and robust statistical methods, may be easy to learn and to apply, other considerations, such as the form of the basic relationships among size, amount of change, and other parameters of interests, may take longer to adapt to. To counter this threat, it will be essential to improve education and increase experience of OD engineers and to provide them with better theories (see Section 4.2). Of course, with a plethora of machine learning and statistical methods and with the large scale and, potentially, confidential data sources, opportunities for malicious Big Lies would multiply. To counter that threat it will be necessary to develop frameworks to reproduce the results on large datasets with potential privacy and confidentiality restrictions.

The third immediate cause for disillusionment will likely be caused by high-profile scandals related to privacy, confidentiality, and identity fraud or related crimes. This would put the limits on the types of data that can be collected and shared. Developing ODS that respect privacy and other cultural and legal norms [56] would help reduce this threat.

4. MEASURE, UNDERSTAND, DO

In this section we consider the defining characteristics of OD and of applications using such data to sketch the engineering principles needed for ODS. In particular, we articulate the need to categorize data quality issues, emphasize the importance of discovering mechanisms that establish relationships among the ODS derived measures and software outcomes, and discuss approaches to integrate ODS with the existing development-support tools.

4.1 Context, Filtering, and Tampering

At a micro level, each recorded event has a context, e.g., what may have been on the actor's conscious and subconscious mind at the time, what tools and practices were used, and why the action was taken. At a more aggregate level, knowledge about the team, project, and application domain are also necessary to place events in context. At large scale, ecosystems, large enterprises, or commercial and OSS hemispheres are all aspects of the context for a particular event. Each event, e.g., a creation of a line of code, a resolution of an issue, a code change, or a business transaction, has a meaning only within that context. For example, the meaning of code churn (added and deleted lines) in a file depends on the programming language used in that file and may not be meaningful in, for example, binary files even if it is possible to calculate a churn number for a binary file. Important context factors are often not recorded in OD.

Traces from operational support tools involve transactions and logs with numbers, dates, developer identities, source code, natural language text, or structured (e.g., xml or html) output. It is not always clear how to convert such OD into observations for a statistical model. Typically it is desirable to start from the atomic observations: individual events or transactions that can be recovered from the traces. For example, in VCS such atomic events are code commits and in ITS they are issue state changes, comments, attachments, or other recoverable events.

Data filtering or tampering involves a variety of intended and unintended manual or automatic actions. First, not all data are recorded, some events may be removed or tampered with later, or a subset selected or changed in other ways for a number of purposes, for example to link multiple data sources.

Using such data to reconstruct what actually happened without a careful consideration of the context and filtering often leads to incorrect conclusions. Any sample of OD events, unfortunately, reflects a combination of multiple contexts (associated with each event) and filters. To address that shortcoming it is necessary to augment data with additional information, often derived from the same or additional OD sources. The most basic approach is to segment the events by context either explicitly or implicitly. An example of the explicit approach would be to impute salient context attributes and use them to filter out undesired context and/or to represent the context in a model. An example of the implicit approach would be to include relevant context factors as nuisance parameters in a model (likelihood). To illustrate this, consider the context of a code change. It involves purpose of the change: to add new functionality, fix issues, or simply to merge different streams of development. The fixes may be done to fix a problem discovered in development, code inspection, testing, or in use. The segmentation approach would augment data explicitly by adding change purpose and problem source. The implicit approach would include change purpose and problem source as nuisance parameters with postulated relationships to the observed values and the model fitting procedure would eliminate them by calculating the marginal distribution.

4.2 What Works in Software Engineering?

ODS can be thought of as precise and, when appropriately calibrated, more accurate measurement instruments that may address some of the longstanding questions of what software engineering techniques are effective and under what circumstances they should be used.

Because of the nature of OD, particular care [34] needs to be taken to draw conclusions based on such data. Some of the confusion with the existing empirical approaches using OD may be traced to the failures to account for factors and relationships that are known to exist. For example, many analyses look at treatment (e.g., use of technology, tool, or practice) effects without adjusting for factors, such as size, that impact virtually any outcome of interest. If the adjustments for size are not made, the resulting differences between the control and treatment primarily reflect the distribution of size and other salient (but not adjusted-for) factors in the control and treatment samples. Such differences between the control and treatment should not be (but usually are) interpreted as being associated with the treatment. To study the effect of a treatment it is crucial to adjust for all factors that are known to affect the outcome. For example, an investigation of the impact of organizational change [37] or work dependencies [8] were considered only after adjustments for numerous other factors that were found to be associated with defects in earlier studies.

Despite such efforts, some of the measures associated with causes of defects are often difficult to obtain. The two particularly important measures are the number of users and the amount of usage. In many domains much of software improvement happens in the hands of users who encounter and report problems and suggest (and sometimes implement) improvements both in Open Source projects [33] and in commercial projects [40]. The number of users and the amount of usage are, therefore, important latent variables that need to be adjusted for when studying defects or requirements. This complex web of interdependencies will have to be uncovered and used to ensure the accuracy of ODS as discussed in Section 5.3.

4.3 How to Act?

To be successful ODS need to address their user needs. ODS addressing strategic software development goals in an enterprise are presented in, for example, [20]. A survey of Microsoft managers and developers [4] provides a list of practical questions for ODS. There will be a need to conduct rigorous requirements elicitation for ODS in other domains and in other organizations.

For a technique or a tool to be used in practice it needs to integrate with existing (or future) operations of a software project or an organization. Most software organizations already have a fair number of mature practices and tools that determine the operation of their software development. The most obvious deployment approach for ODS would be to integrate with the existing practices and tools both to tap data sources and to become a part of the operation.

There are several "entry points" for ODS into day-to-day operations. The current state of practice mandates the use of a minimal set of tools for a normal operation of a software project. VCS keeps track of an evolving code base and its branches. Many projects in open source and industry are increasingly embracing Git. ITS provides workflow for software tasks, such as requirements, user stories, bugs, and, increasingly, other types of tasks. Projects typically use an instance of a continuous build system that allows rapid feedback with the results of compilation and unit tests. More process-oriented organizations may also mandate the use of code inspection, test management, test coverage, and static analysis tools. Other operational support tools include requirements elicitation, project documentation, and requirements tracking. As a result, multiple sources of data can be linked together to be used in ODS and each operational support system provides a potential door to introduce ODS into software development operation.

Software development tools have become extensible platforms simplifying ways to introduce ODS. GUI IDEs provide several paradigms for integration, e.g., Visual Studio provides extension APIs and Eclipse offers APIs and plugins. Issue tracking systems, such as JIRA, are also highly extensible [27], with plugins providing integration with version control, inspection, and static analysis tools. Unlike IDEs or JIRA, the Git version control system has the simplest and the most general unix-like extension API: any Git command, e.g, "git xxx" is looked up as an executable git-xxx. Unfortunately, such extension of individual tools is quite cumbersome and linking data among them may not be trivial. For example, FindBugs provides static analysis errors based on the jar file produced by a build, but the code changes and associated JIRAs are for the source code in a specific branch.

To address some of these data integration problems, Rational Team Concert provides reporting capabilities in a single customizable and extensible package. There are other successful forms of integration as well. For example, Emacs, as the oldest actively developed IDE, not only has an extension language and numerous packages extending its functionality, but, with org-mode and Babel [45], it also has a meta-language that integrates and links together natural language text with various typesetting, computational, and analytic tools as an instance of literate programming [28] and as a platform for reproducible research [46]. For example, Babel allows a user to include the text of a paper, typesetting instructions, and all the needed computations in several programming languages to produce the results presented in the paper within a single text document. This approach of embracing existing tools and languages may be particularly suitable for ODS as it tries to integrate disparate domains and analysis approaches. While being a great tool to publish research results because it provides a complete audit trail for every number presented in the paper, Emacs+Babel does not (yet) have a strong operational support for workflow, parallel work, and construction history.

5. ENGINEERING ODS

Developing ODS is somewhat different from software development for less data-intensive or non-analytic systems. One common use case concerns a workbench involving an analyst creating a report to address a business need or a researcher trying to discover novel phenomena. Another common use case is a back-end tool that processes, augments, or synthesizes data for other tools, such as IDEs, ITSs, an analyst workbench mentioned in the first use case, or some other operational or reporting system. In both cases there is a back-end system collecting and preparing the relevant data from a variety of other tools and data warehouses [12]. The developer of ODS for both scenarios is acting as an analyst and as a researcher. The research part involves understanding peculiarities of the data and basic relationships, while the analyst part involves preparing fixed or customizable reports for the workbench or to feed other tools. In that role, an ODS developer has to have a background in data analysis and also have a nontrivial understanding of the problem domain. The particular type of user interaction, the response time, and other considerations may place additional constraints on the ODS engineer. Data size and limited processing power will require expertise for these areas of software engineering as well. In this section we will be mainly concerned with the engineering topics that are unique to ODS, starting from the specific tools and methods that will be needed to understand the application domain, to establish sound approaches to measure using OD, and to create the building block mechanisms describing how various measures affect each other. Methods to assess and improve data quality, to exploit implicit information, to borrow and evaluate methods from related disciplines, and to validate ODS are discussed next. The section concludes by arguing that operational support tools and, in particular, ITSs, VCSs, and IDEs, should be augmented with ODS capabilities.

5.1 Understanding Application Domain(s)

A nontrivial understanding of the relevant application domain is important for developers of any software because the requirements may not fully convey all details of use scenarios or may be misunderstood without sufficient knowledge of how the software will be used. For ODS it is also necessary to understand how the data came to be. For example, if the ODS will be used to produce reports about user-encountered defects for a software module, then the ODS developer will need to have a basic understanding of how users may be able to report issues, how these reports are handled, and how (and under what circumstances) these reports can be associated with software modules. Examples of how the data comes about include answering a variety of questions, such as were data entered manually or automatically, were there any default values, were data filtered or tampered with in other ways, under what scenarios the data may be missing or removed (cleaned up), and, perhaps, most importantly, was that data important for effective use of the operational support system by the person who was providing it to the system? Data in operational support tools varies in accuracy with some attributes being highly accurate and others being completely unreliable. The attributes that are essential for operation tend to be scrutinized by users of these operational support tools, while the remaining attributes may not be consistently entered.

While this topic may appear primarily as an issue with education and practice, many of the techniques described below can help ODS engineer to answer these questions directly from the available data. Furthermore, some of the techniques from reverse engineering or program comprehension may be suitable for the task of reverse engineering the practices of using operational support tools that serve as sources for the ODS.

5.2 Measures

Perhaps the most important and vexing aspect of measurement based on OD is the fact that virtually every measure may be a result of multiple phenomena, not just of the phenomenon that is being investigated. Disentangling the effects of the "non-primary" or nuisance phenomena is often a formidable task. There are two primary reasons for this difficulty as described in Section 4.1:

- no two events have identical context, so any statistics based on a sample of events represents a mix of multiple contexts (i.e., apples and oranges);

- the filtering mechanism often depends on the value of the measure and the factors that can explain the filtering mechanism are often hard to obtain or unavailable.

For example, modifications to source code reflect what happens in software development because they record the actual act of implementing a requirement or correcting a defect and embody the interactions between the developer and the code they produce. However, as noted in [34], not all modifications to code reflect the act of implementation of new features or corrections. For example, code branching activity primarily concerns creation of development or maintenance streams and merging code from multiple streams. The code changes done in these two contexts are quite different by nature. While it is typically easy to identify (and discard or treat differently) code changes associated with branching, it tends to be more difficult to identify merges. In SCCS, CVS, or SVN, branching and merging are rather tedious activities and tend to be used less often than with the latest generation of VCSs, e.g., Git. Git makes branching and merging both easy and necessary and it is common to see more than 100 branches for a single file [36, 39].

When the number of branches explodes, the usual assumption that code changes are associated with source code creation may no longer hold, because most of the changes may be related to branching and merging. In other words, the same statistic counting the number of changes is more likely in Git to measure primarily the branching activity and is more likely in CVS to measure primarily coding and fixing activity.

As version control tools evolve, the measurements based on them will have to adapt and evolve as well. In addition to branching, Git contains functionality to edit the recorded history either implicitly (e.g., rebase) or explicitly (filter-branch). This requires developing ways to detect and handle such instances of data-tampering in order to retain the integrity of VCS-based measurement systems. Investigation of some of the challenges related to filtering and history tampering in Git repositories is given in [6]. This foundational change in the assumptions of what a code modification is and the associated impact on the extant empirical findings based on change measures obtained from earlier generations of VCSs needs an in-depth investigation.

Another example is focused on code fixes as manifest instances of software defects, a key assumption in much of the research on defect prediction. However, defects discovered in early stages of development (e.g., unit tests) by developers or in code inspection, do not necessarily represent poor software quality. Instead, they may indicate a good development process with issues found in the early stages of development when fixing them is least costly. Predicting such defects, therefore, is unlikely to provide tangible practical value. Customer-reported issues are typically associated with user perception of software quality [40] and are of most concern in commercial projects. Unfortunately, separating customer reported issues in open source projects may not be easy. First, it is not always clear if the issue was reported for a stable release and by end users (not internal developers). Second, the links between code commits and reported issues tend to be tenuous, see for example, [43]. Furthermore, even in commercial projects where customer reported defects are carefully tracked with related code changes, they tend to reflect the extent of usage more than the inherent defectiveness of software as noted in Section 5.3. Finally, the extent of software usage is a critical ingredient needed to interpret measures based on customer-reported problems. As software products are increasingly delivered over the web or via mobile applications, the ability to count the number of users and the amount of usage increases tremendously and should play a big role in software engineering for such domains as discussed in Section 6.2.

Many other units of activity are increasingly being appropriated for measurement by ODS, for example, actions in an IDE, tasks in issue tracking tools, test execution, and various code and information seeking strategies. Just as with software code, changes, and bugs, it will be necessary to assess what each measure means and what phenomena influence it, and to what extent.

As the ODS expand into vast collections of open source or commercial project data, combining or comparing measurements from different projects is a serious challenge yet to be addressed. The code change in projects using CVS is not the same as in projects using Git for reasons outlined above. The scarcity of bugs in one project could simply mean that the software is not used, while in another project it may mean that software has few defects. To compare or summarize even such apparently simple measures as bugs or changes from different projects, it will be necessary to develop methods to address these differences of context.

A common research practice tends to derive the new measures based on convenience. If the measure is easily collected and has a name that appears to match what is needed for the study, it is rare to see a rigorous validation that involves nontrivial understanding on how data came about and what phenomena may have had impact on data being observed or not observed [34, 35]. For ODS to be relevant in practice (and reflect what actually happens), much more work must be devoted to understanding and addressing the concerns listed above: how exactly the data got created, who and when enters values, and under what circumstances, why, and why the data may be missing. With sufficient amount of understanding of what each measure reflects, it will be possible to investigate how general it may be, how to recognize different manifestations of it (e.g, separate a bug fix from new feature), and how to build meaningful models on this more solid foundation.

5.3 Mechanisms

This section provides an illustration of what could be done to establish relationships among various OD-derived measures, such as users, usage, and defects. These relationships can serve as basic laws of software production and are needed to disentangle particularly difficult cases when two or more phenomena interact to produce an event, with one phenomenon acting as a censoring mechanism. A common example of such combined phenomena has the first phenomenon reflecting the presence of a defect in the code and the censoring phenomenon reflecting the chances of that defect being discovered, reported, and fixed. Specifically,

Definition Software defect is an error in coding or logic that causes a program to malfunction or to produce incorrect/unexpected results.

Defects reported by software users are particularly important to software projects as they directly affect users' perception of software. Therefore,

Definition *Bugs* are defects that have been discovered and reported by users and implemented as changes to the source code.

In practice only the combination of the two phenomena (bugs) can be observed. More specifically, for a bug to exist (i.e., for a defect to be observed), there must be a user who has to use the software, encounter and recognize a malfunction or unexpected behavior, and then be willing and able to report it in sufficient detail. Furthermore, the product maintainer has to be interested and able to fix the defect causing that malfunction or misbehavior. Not surprisingly, only a very small and highly biased subset of defects end up meeting the strict definition of bugs. This long chain of events has strong negative feedback loops, because if users experience a lot of problems they may stop using the software altogether or stop reporting issues. As a result, better software (where an individual user has lower chances of experiencing a malfunction) often has more bugs than worse software (where an individual user has higher chances of experiencing a malfunction). This apparent paradox is simply caused by the mismatch of definitions for a defect and a bug (the way a defect is observed in practice). The latter represents a complex equilibrium resulting from actions of different groups of participants in software production: developers, users, support, and sales. For example, users improve software quality by discovering and reporting defects that may be too costly to be discovered otherwise. As new functionality is delivered in major releases, quality conscious users often stay on the sidelines until a second minor release delivers properly working features, bug fixes, and stability improvements. The major releases, being of lower quality, have fewer users and, consequently, fewer bugs. From the statistical perspective the negative interaction between the chances that a defect will be eventually fixed and the number of latent defects means that data are Missing Not at Random (MNAR), see, e.g., [29, 35]. There are no statistical techniques to deal with such cases; the only way to disentangle the two phenomena is to discover and validate mechanisms that explain the relationship between them.

There are numerous related and unrelated paradoxes that can (and will) be explained in a quantitative manner only by discovery and verification of similar laws of software production system. As ODS can not be based purely on definitions but have to rely on collected data, it will be important to realize how the measures commingle several concepts and what could be done to represent various ideal concepts faithfully in such measurement-first approaches.

In particular, the search for such fundamental relationships could be based on a few basic principles. First, consider short-term and recurring relationships because they can be validated more reliably and would be less affected by long term changes in, for example, economy or technology. Second, employ relationships that have a clear mechanism originating from the way software is created and used and constrained by resource and physical limitations of a software projects and individuals. Third, employ relevant additional data sources that may not have been traditionally considered as a part of software development and may be difficult to obtain. Fourth, focus on answering actual software engineering questions because there may be numerous relationships with only a few of them that could help improve software development.

Here we sketch a game-theoretic framework that uses empirically observed relationships in conjunction with the objectives of software developers and consumers to explain the paradox described above. In the simplest conceptual model the two groups of players are developers and users. Developers produce the software and fix problems encountered by users and users derive value from using the software and incur loss when software is not operating properly. For software developers the software provides value through increased usage (directly from licensing revenue or indirectly from the increase in market share). It is typically impossible or not cost effective to ensure that the software does not have any defects for all possible use scenarios. This means that users are an important part of quality improvement process whereby they report bugs that are fixed by developers so that the remaining users have lower chances of encountering a defect-related failure. For simplicity, let's assume that a user i incurs a fixed loss c_i due to a bug they encounter. Let's assume for the moment that the value v_i the software brings to user i is also fixed. We assume existence of a population of users with some distribution on $\frac{v_i}{c_i}$ who may decide to install at any point in time after the software is made available by software developers. Let's denote the probability that a user encounters a bug as $p(t)$ where t is the time elapsed after the software release date. For each user, the strategy to avoid a loss is simple: do not install unless $p(t) < \frac{v_i}{c_i}$. However, $p(t)$ depends on the release itself and on the actions of other users. To model the probability p, let's assume that the chances for a user to encounter a bug decrease as more bugs are reported and fixed. In particular, let $o(t)$ be the number of bugs observed (and fixed) by time t and $p(t) = f(o(t))$, where f is the function that maps observed bugs to the probability that a new user at time t would encounter a bug. Assuming that bugs are only observed at the install date and are fixed immediately, we have

$$p(T) = f\left(\int_0^T n(t)p(t)dt\right), \tag{1}$$

where $n(t)$ is the rate of new users installing a release and $n(t)p(t)$ is the rate of bugs. Furthermore, developers have limited resources and can work only on a limited number of bugs at a time, so they prefer to have the bug inflow rate $n(t)p(t)$ to be a constant $k = n(t)p(t)$. Indeed, the inflow of user-reported issues appears to be relatively constant in practice. This leads to $p(t) = f(kt)$. Empirical observations suggest that f is an exponential function, so $p(t) = e^{-\alpha kt}p(0)$, where $p(0)$ represents the chances that the first user of the release will encounter a bug. The development team can change $p(0)$ by, for example, improving the development process and by increased testing and alpha/beta trials. Developer value comes from more users and the number of users at the time T after the release date can be obtained via

$$\int_0^T n(t)dt = \int_0^T k/p(0)e^{\alpha kt}dt = \frac{1}{\alpha p(0)}(e^{\alpha kT} - 1), \qquad (2)$$

thus having a smaller $p(0)$ (chances of failure at release date) and higher k (rate at which bugs can be fixed) would lead to more users and, consequently, provide more value to developers. This assumes, of course, that the population of users who have $p(t) < \frac{v_i}{c_i}$ is not depleted.

Given that the chances of failure decrease over time, each user would benefit by waiting longer to maximize $v_i - c_i p(t)$. This implies that the optimal strategy for each user is to wait as long as possible before installing a release — a lack of Nash equilibrium. Equilibria are represented by points in a player strategy space where a single player can not benefit by changing their strategy given fixed strategies of the remaining players.

However, the assumption that v_i is fixed is unreasonable. More likely, the incremental value brought by the software depends on how long it has been used, so v_i is also a function of time: $v_i(t - t^i_{inst})$ where t^i_{inst} is the time at which the user i installs software. Assuming a linear relationship we get $v_i(t - t^i_{inst}) = (t - t^i_{inst})V_i$. Now the condition $p(t) < \frac{v_i}{c_i}$ can be rewritten as $p(t^i_{inst}) < (T - t^i_{inst})\frac{V_i}{c_i}$. The left side can be rewritten as $p(0)e^{-\alpha kt^i_{inst}}$. The install date t^i_{inst} maximizing the expected value would be $\frac{1}{\alpha k}\ln \alpha kp(0)\frac{c_i}{V_i}$ if $\alpha p(0)c_i/V_i > 1$ and zero otherwise. If the software quality is high and it has value for users, the best strategy for a user is to install right away ($t^i_{inst} = 0$). If the value V_i is zero, the best strategy is not to install. High cost of failure c_i and high initial probability of failure $p(0)$ increase the installation delay for the optimal user strategy.

The total number of user-reported bugs under this simplistic model depends only on the duration of time after the release date. Despite the model's simplicity, it can provide an explanation for why major releases tend to have fewer users and why minor releases may have a similar number of bugs reported per unit time as major releases, but the chances for a user to encounter a bug are much lower than for major releases. The model relies on three mechanisms: bugs being discovered with more users, increase in software quality with more bugs being encountered and fixed, and constraints on the productivity of the development team with a preference for a constant inflow of work.

Remarkably, the model provides the elusive relationship between the quality of software at launch $p(0)$ and the revenue expressed as the number of users shown in Equation 2.

5.4 Assessing Data Quality

Software quality is an important aspect of software engineering and data quality is an even more critical concern in ODS. When producing ODS we want to know when data are not good enough for a task at hand. For example, data may not have any relevant information, it may not be possible to disentangle the events of interest from other commingled data, or the data may have been tampered with to the extent of being no longer usable.

Traces from each tool may have specific issues for an ODS developer to be aware of. As noted above, any VCS may have branching or administrative changes, and Git may have a modified history. In issue tracking tools many of the fields are not used in practice, so it is not likely that data collected from such fields will contain any relevant information. Eliminating such fields would help reduce the dimension of data for later analysis. Even when an attribute is used in operation it is important to understand how it is populated, by whom, and how important is its accuracy for efficient operation. For example, some attributes may have default values, strongly affecting the frequency distribution. Some attributes may be modified or seen only by a particular role-task combination. For example, an external bug reporter may only modify their own bugs, or a customer may only be able to see their own issues. The same attribute may be used differently in different projects or by different roles. Even when the purpose of an attribute is similar, the exact definition and the practices of using it may vary among projects and individuals. Some attributes, such as issue priority may not affect operational decisions as much as, for example, a problem description.

Pieces of data that are used to link different tools need to be subject to a particular scrutiny. For example, the link between issues in ITS and VCS can be established via issue ID, but, especially in open source projects, the issue ID may be missing. If the chances of a link being missing depend on the response variable in the analysis (a likely occurrence when counting fixes related to issues) we can not expect the linked sub-population to be representative of the entire population. We may, therefore, need to establish missingness mechanisms discussed in Section 5.3.

Looking at long-term repositories poses challenges of their own. People change names, old accounts are appropriated by other individuals, and data gets imported from older tools, for example, many commercial and open source projects have imported their CVS or SCCS repositories to Subversion and then to Git. The new tools may not accommodate all the features of the old tools and are often used in a different manner. ITS tools get merged and cleaned over time, in effect tampering with their history.

Identification of individuals is particularly important for a substantial part of OD-based inference, yet it is difficult to do, especially over an extended period of time or across several operational support tools. Typical approaches, e.g., using an email or a login as identification are not reliable and require extensive validation because the same email may be used by multiple individuals and be one among multiple emails used by a single individual. Even in human resource tools, the individuals' IDs often change as they move from one country to another. A person may spell their name differently and may change their legal name as well. Only by combining multiple sources of information, such as email, login, name, organization and address from multiple snapshots of the organizational data it is possible to have a reliable identifier for an individual. Identifiers for an individual often vary among different tools, For example, IDs in SalesForce, VCS, and an issue tracker may be different for the same individual. In open source, or in cases where data from multiple organizations are combined, we may need to rely on the activity patterns or on the writing and coding style to establish the identity of the individual.

This laundry-list of potentially serious data quality issues is often either ignored or treated on a case-by-case basis in much of the extant work. More research is needed to find effective ways to detect and mitigate these issues to ensure the integrity of ODS and to make engineering ODS more efficient.

5.5 Implicit information

A significant amount of information can be obtained not from what is recorded in the operational support tools, but from physical or resource constraints defined by the way these tools are used. For example, each person has an upper limit of what they can accomplish in one day or in one month, even though there may be substantial variations among individuals. A massive commit activity recorded in one day by a single individual must mean that either it results from the effort spent over preceding days or months or it represents an an action that was not effort intensive, for example to merge or to import code.

The limited resources of the support team require products to control the inflow of user issues in order to be able to resolve them rapidly or risk alienating users. It is thus reasonable to expect that the inflow would stay relatively constant in an effective organization.

By assuming that an engineer spends approximately one-person-month of effort per month (in reality, engineers may take some of the days off or work overtime), it is possible to determine relative effort spent on different types of tasks even if the tasks overlap or do not align with month boundaries [19].

Identifying and exploiting such implicit information based on physical or resource constraints may prove to be a rich research area and play a more prominent role in the future.

5.6 Statistics, Machine Learning, Optimization

There is much to learn and borrow from statistics, machine learning, optimization, and game theory in software and ODS engineering. However, these disciplines did not evolve to serve software engineering applications. While each discipline has a lot to offer to software engineering, almost all methods and tools need to be applied with caution, as commonly held assumptions (determined by the target areas of each discipline) tend not to hold in the software engineering context. A typical example of such problem is given in Section 5.3 — the chances of observing a defect are strongly related to the number of bugs. Statistics does not consider such problems, because there are no general methods to solve them, the only way to address them is to find mechanisms within the software engineering domain. Other disciplines require addressing such problems as a prerequisite for the analysis. Some of the mechanisms in software engineering, such as the dependence on size, are essential for almost all models, and excluding size predictors is rarely advisable.

Other notable distinction of software engineering is that distributions are virtually never normal, with most metrics being highly correlated (in large part because each represents a slightly different combination of the same set of phenomena). As observed in, e.g., COCOMO [7], the models are typically multiplicative, not additive. A logarithmic transformation is, therefore, needed before fitting a model. As an added benefit, the logarithmic transformation may reduce the high skew often observed in software engineering data.

For a statistical approach an important goal is to create a plausible model (simplification) of the phenomenon and to evaluate various hypotheses, or, in other words, to understand the phenomenon. Machine learning (ML) is more geared toward making good operational decisions, such as recommending information or actions, without necessarily concerning itself with the understanding. These differences are somewhat superficial, as statistical models can also be used for prediction and some ML techniques are based or derived from such models. Most ML approaches, however, give no insight about why the suggested decision was chosen. The lack of transparency on how the prediction is made may present a serious roadblock for strategic decisions that need clear reasoning and management support.

In software engineering publications it is not unusual to see a prediction method that involves complicated procedures that may require a PhD to apply it and that are evaluated entirely based on their accuracy. It may be unreasonable to expect that such methods could ever be implemented and maintained in an operational support tool or used by an industry analyst. Furthermore, the accuracy evaluations should take into account a realistic user scenario to make it clear how the proposed methods would bring value in practice. For example, in many software development decisions the utility/loss function is highly asymmetric requiring either a high precision or a high recall, and that trade-off varies among projects. In other scenarios, high accuracy may not be critical, but the simplicity of deploying the tool would be far more important. In the future, the analytic techniques will probably have to rely on models that make the results more transparent and more actionable. To apply such techniques appropriately, more research is needed to understand trade-offs among precision, recall, and cost of doing the prediction (the cost of operating and maintaining ODS).

As economic incentives and social interactions in software development are increasingly being modeled, we can expect to see more game-theoretic models in the future. To build custom models for ODS, for example, to solve disaggregation problems, relevant techniques from the optimization domain are likely to be brought in.

5.7 Validating ODS

The task of validating ODS appears to be virtually impossible. ODS use complex, noisy, and changing data with no “correct” result to use as a gold standard. The only way to assure validity of the results is by ensuring that every step in the processing chain is valid and appropriate for the type of analysis performed and for the type of data being used.

It is a hierarchical task starting from a careful inspection to understand the provenance and distribution of each piece of data, of the problem domain, and of the practices used. In cases when assumptions need to be made about augmenting the data, a suitably selected stratified sample needs to be verified with an independent data source, e.g., interviews.

Relationships among attributes need to be investigated to decide how to select an uncorrelated subset of variables for a subsequent analysis.

It is important to realize that values in the operational support tools may be inaccurate. For example, the classification of issues as defects or a priority assigned to an issue are often inaccurate. A sensitivity analysis may need to be performed to gauge the amount of uncertainty in the results with respect to the inaccuracies in these values. Such evaluations are rarely performed in software engineering [47] and more research is needed to investigate the extent of inaccuracy in various operational support tools. Given the recent attention to poor reproducibility [25] and analytic results suggesting that traditional cross-validation and bootstrap lead to serious over-fitting [63], it is likely that more work will appear in the area of sensitivity to data errors.

While it is not a focus of this paper, it is important to recognize that there are many validation techniques in, for example, scientific computing, databases, and software engineering itself that could be adopted to validate ODS.

5.8 How Might the Future ODS Appear?

Widely used packages, such as R, are targeted to a researcher or an analyst. The main beneficiaries of ODS in software development will not be analysts or researchers, but developers, testers, product managers, and other project participants. The learning curve and the deployment hurdles would be lowered if ODS were to make their existing tools that are necessary for daily operation and reporting smarter, e.g., with "OD inside". In fact, such ODS are hiding in plain view in the form of VCS, ITS, build, test, and other software tools. It is hard to imagine present software development without these tools, yet each presents an instructive, if somewhat rudimentary, ODS. The basic functions of such tools involve the ability to support operations, ensure audit trail, and to provide some reporting capabilities. The operational support involves, e.g., workflow in ITSs, code modification, branching, and merging in VCSs, building executable code and running tests in build tools. Audit trail is represented as an issue activity history in an ITS or a version history in a VCS. Reporting capabilities include issue backlog charts in an ITS, a list of past changes in a VCS, or a history of past builds and tests. The information from these tools helps developers to make appropriate decisions (e.g., fix a build problem) and reporting capabilities allow analysis and prediction (e.g., the size of issue backlog helps to determine project's schedule). Some of the existing tools, e.g., Sonar [52], are focused mainly on reporting capabilities by showing various statistics about the code. Even though they are instances of somewhat sophisticated ODS, the lack of integration with the development workflow may make them less effective in practice. A tighter integration of such ODS with development workflow would make it easier to encourage and measure their use without the need to introduce additional process requirements.

Presently, many of development-support tools (with some of them being ODS) are integrated into an IDE, because developers prefer to see the code they modify as a context for any action they contemplate and because the modifications to code are how source code creation and maintenance proceed. However, developers are typically focused on their primary task of coding when using an IDE and may not engage ODS. Even if the ODS is engaged from an IDE and a worthy task is identified, developers may not have time to implement it immediately. Having capabilities to create and track a task associated with ODS recommendations would simplify adoption of such recommenders. These recommended tasks could then be prioritized, resources allocated, and implementation scheduled.

Task tracking is pervasive and presents an excellent source of data and a target for improvement by ODS. For example, companies often have tools that track all software lifecycle stages and tasks, such as, requirements, user stories, inspections, static analysis issues, tests and test execution, customer deployments and experiences. This creates a possibility to link information from these disparate domains and tools.

Many of the topics discussed above directly or indirectly bear on the design decisions facing ODS and on the operational support software providing digital crumbs to feed ODS. In particular, the need to support operations, ensure audit trail, provide reporting, and easy (and early) detection and recovery from the inevitable bugs is as essential for ODS as for any other operational support tool.

In summary, there are successful ODS that are already used in software engineering, such as task tracking. They provide operational and reporting (analytic) capabilities and could be expanded by additional OD-based capabilities or emulated in other ODS targeting IDEs, VCSs, or build and test tools.

6. IMMEDIATE CHALLENGES

Below are a few examples of trends that appear to stretch the capabilities of existing development tools and procedures and are likely to benefit from novel types of ODS.

6.1 Drowning in Code

The amount of source code grows for a variety of reasons. Active developers write code over time, thus even if the number of developers in the world stayed fixed, the amount of code would continue to increase. There are more developers over time as software engineering is transitioned to numerous locations over the world in search of inexpensive talent. A modern VCS, such as Git [59], not only provides powerful branch and merge tools, but also makes creation of branches and additional repositories a necessity, thus resulting in a proliferation of branches and repositories. IDEs and code generation tools replicate innumerable templates that are then maintained by changing code generation parameters or by manual editing. The need to support multiple platforms, especially for mobile applications, also contributes to the amount of branching. In the enterprise, the code proliferates as the original development teams and culture of maintaining lean code are replaced by contracted work with less emphasis on ensuring that only the minimal amount of code is used. The evolution of technology standards, for example, the evolution of telephony protocols from analog, to TDM, H323, SIP, and now to WebRTC, requires new code to implement a similar functionality.

So far, the experimental evidence [49, 50] suggests that the amount of code is the primary driver of maintenance effort. If the amount of code continues to grow, the projects may become unmaintainable. In particular, just being able to sift through this mass of code in various repositories and branches to find what is needed or relevant will become difficult. The following areas are likely to benefit from novel ODS designed to cope with the abundance of code.

First, navigating among hundreds of branches would require more sophisticated diff and merge tools. Tools to summarize and suggest the most relevant branches and ways to create diffs and merges with hundreds or thousands of versions to support certain development scenarios, for example, a multi-platform development, are likely to emerge. We will probably see some applications of text analysis techniques such as statistical translation both being trained on such diverged code bases, and also being used to merge them permanently or on-demand. Other techniques, such as topic analysis [24], may help with categorization and a search for relevant functionality. We will see more procedures and tools to support various branching strategies [48], both within an enterprise, among companies, and in open source. In addition to lessons from the Linux kernel development [17], we will see examples of more radical branching in the mobile platforms. Presently the user interface is implemented and tuned to each platform completely independently, often by a different development team. Merging feature enhancements and fixes among these independent implementations will be a worthy challenge for ODS.

Various ways to identify and reduce code-bloat will be needed. In addition to code analysis, change analysis strategies will appear that identify under what circumstances (and by whom) code-bloat is introduced and suggest ways to reduce it.

6.2 Just-in-Time Features

Web services and mobile applications offer perfect opportunities to apply OD-based approaches because the data collection is centralized, the usage data are more readily available, and many of the software engineering problems are new, for example, end users participate more fully in the quality and feature improvement activities. Software engineering innovations offer clear competitive advantages in both industries, so the state-of-the art requirements elicitation, testing, deployment, and measurement of customer satisfaction are likely to be guarded as commercial secrets. An industry based on a business model to provide analytics for software makers in web service and in mobile space has grown. Not only the business aspects (such as in-app-purchases) are addressed, but full mobile application performance frameworks are also provided. They may contain instrumentation for the mobile application and the analysis of the performance data (e.g., crashes or memory usage) from users who install such instrumented apps. For mobile applications and web services, estimating usage is not as complicated as for packaged or system software, and user experience (and resulting business benefits) could be assessed through suitable techniques that involve both, instrumenting client and server software, and devising ways to clean and model the resulting data.

The ability to see how the modifications to existing features or new features convert into usage patterns and cash (either through downloads, advertisements, or in-app purchases), completes the business loop of mobile application development. The full development cycle could be completed within hours, so it is not unreasonable to expect ODS that experiment with adding or removing features of the mobile application to maximize user experience, future cash flows, or some other objective.

6.3 Project Universe and Parallel Evolution

Open source projects provide source code for anyone to see and use. Often, not simply the code, but version control and issue tracking tools allow public access as well. There are numerous software projects with a public access to their VCSs. For example, several million projects are on github.com and more than 300K on each sourceforge.net and code.google.com. Many software organizations are centralizing the development resources within a corporate cloud as well. With network bandwidth becoming less costly, it does not make economic sense for each project to support their own version control, issue tracking, inspection, build, and numerous other tools. The open source and corporate clouds now host massive collections of software development data in a variety of support tools. This provides both a data source and an outlet for new ODS to emerge. Having vast collections of projects poses both challenges and opportunities. The simplest opportunity is to conduct a basic census: what is the distribution of code, people, and their activities and how does it depend on context? Higher level questions would include measurement and comparison of innovative practices and artifacts. The answers to such questions may provide a basis for engineering socio-technical systems with desired features, for example, longevity or efficiency. However, the lack of methods to segment projects by salient context factors and methods to integrate non-code data, such as developer identity and issue history, are two formidable obstacles that make progress difficult in this area.

Experiments involving construction of two or more software systems with identical functionality could provide unequivocal answers about what software development methods are effective, but such experiments are quite expensive even for small systems [1]. Fortunately, large software project collections may be able to provide a less expensive alternative. Many of the open source projects have been forked or developed similar functionality independently. As companies merge, they often acquire competitors with products that have almost identical functionality but have been developed completely independently over many years, with each using the development and support practices of the originating company. Such natural experiments [13] may show if the differences in software development practices are associated with development effort or software quality.

6.4 Data Quality in Operational Support Tools

Data accuracy is not simply an issue for validating ODS (see Section 5.7). Much of the operation of a software development (or other knowledge intensive) organization is based on the business process that is driven by data in the operational support tools. The effectiveness of every organization, therefore, depends on the accuracy of data it uses to operate. In any organization of nontrivial size, work consists of tasks that flow among the participants according to a business process or a common practice. The flow and the decisions made by each participant critically depend on the information in the supporting system. For example, an incorrect severity for a task may waste resources on less urgent issues while leaving more urgent tasks unattended. A customer issue assigned to a wrong product team takes much longer to resolve (negatively affecting user-perceived quality) and wastes developer effort. Being able to estimate the accuracy of data that is used to support operation (e.g., to route issue workflow) can improve the effectiveness of the software development by re-routing the work based on the amount of uncertainty associated with fields that are important for operational effectiveness. This can be done, for example, by adjusting the workflow for tasks with likely-erroneous data to improve the quality of data before taking the operational action, e.g., sending the issue to a specific product development team or marshaling resources for a particularly urgent customer issue [62].

The intriguing aspect of employing ODS for data quality improvement in operational support tools is the recursive nature of the process — as data quality is improved with ODS it becomes easier to implement other ODS based on this, now less-problematic, data. As discussed in Section 5.8, the operational support tools will be increasingly incorporating elements of ODS, and designing such self-cleaning operational support tools will likely attract its fare of research attention. Some instances of self-cleaning operational support tools already exist in the form of sophisticated moderation and meta-moderation schemes used by online communities, e.g., on slashdot.org [42]. Such schemes typically rely entirely on the manual input from community participants, and it is likely to be an area where a suitably designed ODS could help to reduce the amount of effort involved or to increase the accuracy of reputation and other data.

7. A BACKWARD LOOK FORWARD

Feedback from users in various environments will be needed to understand their OD needs [4]. However, such feedback is limited by the existing constraints and tools available to informers. It may, therefore, be instructive to contemplate what software development may look like in the near future to better understand and cater to potentially emerging needs. The next scenario of software call center, where a software task is specified, designed, implemented, verified, and deployed, all within the span of a single call, is highly speculative, but is well within the realm of what is possible. In particular, the limiting factor and the goal of the following scenario is the extremely compressed response time as may be needed to implement new mobile features discussed in Section 6.2.

The dream of low-cost just-in-time software development has not been realized yet, and many alleged silver bullets, e.g., focusing on software process, have not delivered as much as was expected. There are many formidable obstacles to more predictable software development. For example, software design and maintenance are creative activities that rely on tacit knowledge well concealed in developers' minds, and on the often peculiar organizational culture of teams creating software. At the same time, as more aspects of human activity are supported by software, there is a practical need to make software development more transparent, predictable, and controllable. Ignoring that need is not a wise choice despite the perils and challenges that await someone trying to address it. ODS are already offering a possibility to address that need by making software development more transparent [11].

An instructive example of knowledge work that was simplified and templatized is that of a call center agent. Many customer support functions, while inherently complex and open-ended, have been implemented in a present-day call center. In essence, such workers interface with multiple business tools that can not (or are too expensive to) be integrated or can not be exposed to the public to help customers with the less common tasks that are not resolvable via an interactive voice response (IVR) system or via the customer support web site.

The agents need to take a fairly short (one to few days) training course to be familiarized with the customer interaction scenarios and with the screens of the various tools that may be needed in common customer-support tasks, e.g., to check if the payment was received but was not yet applied to the account. An important part of the job is to recognize the circumstances where expert help is needed and direct the customer (or issue workflow) accordingly. We will argue that the work of a software engineer has a number of similarities to the work of the call center agent. Both are knowledge workers responding to customer issues, obtaining and modifying data, contacting other agents or experts, and opening issues to be tracked. In development as in a call center, much of the time is spent in clarifying the requirements or reproducing the problems as well as in finding relevant expertise and waiting for the responses from experts.

A radical difference between developers and call agents is the duration of their tasks. Software development tasks are not resolved in real-time but are tracked as issues in ITS or email exchanges and may take days or weeks to get resolved in contrast to agent tasks that are resolved in minutes. The development task tracking tends to slow the work substantially and call centers could provide an example of a workflow that could speed up some aspects of software development. Would it be possible for an a already skilled developer with some project training to resolve a substantial portion of tasks rapidly (within a few minutes)? Such developers would work, as in the case of call center, under supervision and escalate more complicated tasks to domain experts. Below are possible scenarios on how such a feat might be accomplished.

One of the most central parts of the call center is a reporting system. It is an instance of ODS that has been in operation for decades. It provides both real-time and historic reports. Real-time reports address urgent issues emerging in operations of a call center: unusual inflow of calls, need to assign more agents with relevant expertise, and management of waiting time. Historic reports are used to determine the quality of customer experience and performance of agents and supervisors. They include reports that determine agent compensation and reports that detect cheating behaviors. This is exactly the domain of OD that has not yet seen a wide adoption in software development. Analogous measurement techniques would be useful in, for example, outsourcing software development work.

Many of the call-center-like tools for routing work, tracking, and reporting are already deployed or exist as prototypes in software development, but the integration necessary to create such a software production analog of a call center is still lacking. This conceptual operation would use traditional issue tracking tools to route work and evaluate work progress, code-inspection-like tools should be built to supervise the quality of work, code and task expertise tools would be used to measure expertise and route escalations and tasks accordingly, and mentorship tools would assign supervisors and mentors. ODS would play a role associating individuals and code, determining the importance and relevance of the tasks, and assuring (and improving) accuracy of the information. Mentorship analysis would help to construct expertise networks, risk detection techniques would identify the riskiest areas of the code, most important tasks, and customers with the most urgent needs.

While some aspects of software call center may require a lot of work to have an industrial deployment, something not entirely unlike it already exists in practice. In a multitude of on-line marketplaces there are some mature companies with a long history that are targeting software development. Such companies, for example, [61] provide a marketplace for independent contractors to find work and for companies to outsource small projects. Projects can be very small, with some projects having a minimum price of US$200 with a US$10 hourly rate (guru.com on January, 2014) These marketplaces do not provide a real-time response and are lacking a network of experts for these freelancers to rely on in case they encounter an issue. The mentor network for such marketplaces could, however, be constructed based on the approach used in github.com, which, along with development-tool-hosting service, provides a rich social networking functionality such as feeds, collaborators, and watchers to make developer work and interactions more transparent. Many challenges of sharing compensation, routing escalations, and ensuring trust would need to be addressed to make such more effective marketplaces function.

Initially, therefore, software call centers would probably focus on expanding basic customer support within an enterprise, and the Information Technology (IT) support is probably the most likely initial deployment environment. Perhaps a gradual move from mundane IT support tasks to more innovation oriented professional services tasks could be an early testbed of such just-in-time software development. Over time, however, the biggest impact may be in outsourcing and contract work, where the rapid training scripts and objective measures of performance are still sorely lacking.

It is worth noting that the population of developers isn't large and many individuals may not have a desire or an aptitude to write code. It is important to recognize, though, that writing code is but a relatively small and shrinking part of software development, because a substantial and increasing portion of work involves configuring software, composing software from preexisting blocks (mashups), or simply reporting and triaging issues and requesting additional features. ODS can certainly help with these tasks both by providing actors with the information needed as described in [53]. To make agents more efficient, call centers typically collect all the relevant information about the customer and the task (typically via an IVR system) and present it to the agent at the time of the call. Similarly, even the most expert developers would be more productive if all the information salient to the decision at hand would be proactively and unobtrusively available.

8. SUMMARY OF THE ROADMAP

To satisfy the growing need for ODS in software engineering and other fields and to avoid compromising the integrity of the results produced by these systems, it will be necessary to develop both basic principles and tools that allow effective engineering of ODS. This research may take the form of systematizing existing best practices, of borrowing relevant approaches from other domains, e.g., databases, and of discovering the principles de novo for ODS-specific challenges.

We discuss defining features of ODS to guide the work of building the engineering principles:

- no two events have exactly the same context;
- data are usually incomplete;
- data are usually incorrect, filtered, or tampered with.

Basic OD measurement principles are defined by the necessity to recognize typically unavailable context information and take a variety of steps to segment the events.

Because OD do not capture all actions of interest and the chances that an action would be captured in OD often depend on an entity that is being estimated, it is necessary to develop a library of basic mechanisms describing the relationships among entities for the software engineering domain (or for the domain the ODS are targeted to). Such mechanisms could then be incorporated into models and used to segment by context, impute missing values, and identify and correct wrong, filtered, or tampered with data.

More research is needed to understanding fundamental constraints faced by individuals, groups, and organizations. Such constraints can serve as implicit information used to estimate quantities that are difficult to obtain, have unreliable data, or are simply not available.

Particular areas that need immediate research attention are related to data quality issues, such as effective methods to identify data entry problems, clean data, augment or segment events, and develop robust methods to establish subject identities.

Statistics, machine learning, operations research, and game theory have a significant role to play, but particular care needs to be taken when applying these methods in ODS because the assumptions taken for granted in many of the borrowed techniques do not apply for OD in general and for software engineering in particular. More research is needed to understand what assumptions used in these disciplines do apply in OD.

ODS are software systems that need maintenance and validation. These concerns are important when engineering ODS, but they have yet to be thoroughly investigated.

We discussed two high-level use cases: analyst's workbench and an augmentation of an existing operational support system and argued for the need to consider the later model to achieve a wider acceptance in software engineering practice. In particular, issue trackers appear to be a good target for the augmentation approach.

In addition to the urgent need to develop engineering principles for ODS, we discuss several emerging applications domains where approaches relying on OD may be particularly suitable. Modern VCSs make it easy and necessary to create branches and increasing amounts of code are created and accessible publicly or within enterprise. This offers opportunities for, for example, reuse of code and of code fixes, but it also makes it difficult to navigate among hundreds of branches to find the relevant code or fix.

Large collections of public and enterprise project repositories provide tantalizing opportunities to answer some of the most vexing questions in software engineering. For example, it is cost prohibitive to run experiments that implement the same functionality in multiple large software projects to evaluate effectiveness of various software development methods. However, a large collection of projects is likely to contain multiple projects that happen to implement a similar functionality. For example, an enterprise may acquire a competitor with a set of similar software products. Such matched groups of products could then be studied using a framework of natural experiments [13] embraced by archival studies in economics and epidemiology.

Mobile application software provides opportunities for unprecedentedly rapid updates of software for end users, but the highly fragmented client platforms make it difficult to assure software quality. Fortunately, it is feasible to collect application deployment, usage, and performance information in these domains. ODS based on such data could provide capabilities to fully utilize the rapid update cycle without compromising user experience.

We also raise questions about the limits of software engineering and the research needed to go beyond them by describing a concept of a software call center — where the software issue is reported, reproduced, fix designed, implemented, and delivered, and all of this happens within the duration of a single call.

The necessity to measure the context of software development, the blurring of software development boundaries, and the explosion of measurement in other areas of human endeavor represent three powerful and converging trends that will redefine key notions of software engineering and practice. For example, the fact that users and usage are important contributors to software quality is already partially utilized in crash reporters, user support groups, and in the deployment strategies of web services. However, just-in-time patches fixing defects in real time or just-in-time new functionality defined and delivered on-demand, could become a reality as the software development, usage, and measurement become more tightly integrated.

9. ACKNOWLEGEMENTS

I'd like to thank Jon Bentely, John Palframan, Randy Hackbarth, and Sriram Rajamani for their valuable comments.

10. REFERENCES

[1] Bente C.D. Anda, Dag I.K. Sjøberg, and Audris Mockus. Variability and reproducibility in software engineering: A study of four companies that developed the same system. *IEEE Transactions on Software Engineering*, 35(3), May/June 2009.

[2] V.R. Basili, R.W. Selby, and D.H. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, pages 758–773, July 1986.

[3] V.R. Basili and D.M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, 10(6):728–737, 1984.

[4] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *ICSE*, Hyderabad, India, June 2014. IEEE CS.

[5] L. A. Belady and M. M. Lehman. Programming system dynamics, or the meta-dynamics of systems in maintenance and growth. Technical report, IBM Thomas J. Watson Research Center, 1971.

[6] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining git. *2013 10th Working Conference on Mining Software Repositories (MSR)*, 0:1–10, 2009.

[7] B.W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[8] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. Software dependencies, the structure of work dependencies and their impact on failures. *IEEE Transactions on Software Engineering*, 2009.

[9] Per Cedeqvist and et al. *CVS Manual*. May be found on: `http://www.cvshome.org/CVS/`.

[10] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Subversion Manual*. May be found on: `http://svnbook.red-bean.com/`.

[11] Laura Dabbish, H. Colleen Stuart, Jason Tsay, and James D. Herbsleb. Leveraging transparency. *IEEE Software*, 30(1):37–43, 2013.

[12] Data warehouse. `http://en.wikipedia.org/wiki/Data_warehouse`.

[13] T. Dunning. *Natural Experiments in the Social Sciences: A Design-Based Approach*. Cambridge University Press, 2012.

[14] S.G. Eick, J.L. Steffen, and Sumner E.E. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957 – 968, November 1992.

[15] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *ICSM*, pages 190–197, 1998.

[16] Hype cycles. `http://www.gartner.com/technology/research/methodologies/hype-cycle.jsp`.

[17] Linux kernel. `http://en.wikipedia.org/wiki/Linux_kernel#Development_model`.

[18] R Grady and E Caswell. *Software metrics.* Prentice-Hall, Englewood Cliff, 1987.

[19] T. Graves and A. Mockus. Identifying productivity drivers by modeling work units using partial data. *Technometrics*, 43(2):168–179, May 2001.

[20] Randy Hackbarth, Audris Mockus, John Palframan, and David Weiss. Assessing the state of software in a large enterprise. *Journal of Empirical Software Engineering*, 10(3):219–249, 2010.

[21] M. H. Halstead. *Elements of Software Science.* Elsevier – North Holland, 1979.

[22] Ahmed E. Hassan, Abram Hindle, Per Runeson, Martin Shepperd, Premkumar T. Devanbu, and Sunghun Kim. Roundtable: What's next in software analytics. *IEEE Software*, 30(4):53–56, 2013.

[23] Ahmed E. Hassan, Richard C. Holt, and Audris Mockus. Report on MSR 2004: International workshop on mining software repositories. In *ACM SIGSOFT Software Engineering Notes*, 2005.

[24] Abram Hindle, Neil A. Ernst, Michael W. Godfrey, and John Mylopoulos. Automated topic naming to support cross-project analysis of software maintenance activities. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 163–172, New York, NY, USA, 2011. ACM.

[25] John P. A. Ioannidis. Why most published research findings are false. *PLoS Med*, 2(8):e124, August 30 2005.

[26] J. Jelinski and P. B. Moranda. Software reliability research. In W. Freiberger, editor, *Probabilistic Models for Software*, pages 485–502. Academic Press, 1972.

[27] Jira plugins. `https://marketplace.atlassian.com/plugins`.

[28] Donald E. Knuth. *Literate Programming*. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[29] R. J. A. Little and D. B. Rubin. *Statistical Analysis*

with Missing Data. Willey Series in Probability and Mathematical Statistics. John Willey & Sons, 1987.

[30] T. J. McCabe. A complexity measure. *IEEE Trans. on Software Engineering*, 2(4):308–320, Dec. 1976.

[31] Katina Michael and Keith W. Miller. Big data: New opportunities and new challenges [guest editors' introduction]. *Computer*, 46(6):22–24, 2013.

[32] Anil K. Midha. Software configuration management for the 21st century. *Bell Labs Technical Journal*, 2(1), Winter 1997.

[33] A. Mockus, R. F. Fielding, and J. Herbsleb. A case study of open source development: The Apache server. In *22nd International Conference on Software Engineering*, pages 263–272, Limerick, Ireland, June 4-11 2000.

[34] Audris Mockus. Software support tools and experimental work. In V Basili and et al, editors, *Empirical Software Engineering Issues: Critical Assessments and Future Directions*, volume LNCS 4336, pages 91–99. Springer, 2007.

[35] Audris Mockus. Missing data in software engineering. In J. Singer et al., editor, *Guide to Advanced Empirical Software Engineering*, pages 185–200. Springer-Verlag, 2008.

[36] Audris Mockus. Amassing and indexing a large sample of version control systems: towards the census of public source code history. In *6th IEEE Working Conference on Mining Software Repositories*, May 16–17 2009.

[37] Audris Mockus. Organizational volatility and its effects on software defects. In *ACM SIGSOFT / FSE*, pages 117–126, Santa Fe, New Mexico, November 7–11 2010.

[38] Audris Mockus, Todd L. Graves, and Alan F. Karr. Modelling software changes. In C.E. Minder and H. Friedl, editors, *Good Statistical Practice*, pages 175–179. Austrian Statistical Society, Wien, Austria, July 1997. Proceedings of the 12th International Workshop on Statistical Modeling, Biel/Bienne.

[39] Audris Mockus, Randy Hackbarth, and John Palframan. Risky files: An approach to focus quality improvement effort. In *9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2013.

[40] Audris Mockus and David Weiss. Interval quality: Relating customer-perceived quality to process quality. In *2008 International Conference on Software Engineering*, pages 733–740, Leipzig, Germany, May 10–18 2008. ACM Press.

[41] 11th working conference on mining software repositories. `http://2014.msrconf.org/`.

[42] Nathaniel Poor. Mechanisms of an online public sphere: The website slashdot. *Journal of Computer-Mediated Communication*, 10(2), 2005.

[43] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar T. Devanbu. Sample size vs. bias in defect prediction. In *ESEC/SIGSOFT FSE*, pages 147–157, 2013.

[44] M.J. Rochkind. The source code control system. *IEEE Trans. on Software Engineering*, 1(4):364–370, 1975.

[45] Eric Schulte, Dan Davison, Thomas Dye, and Carsten Dominik. A multi-language computing environment for literate programming and reproducible research. *Journal of Statistical Software*, 46(3):1–24, 1 2012.

[46] Matthias Schwab, Martin Karrenbach, and Jon Claerbout. Making scientific computations reproducible. In *Computing in Science & Engineering*, pages 61–67, 1997.

[47] Martin J. Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. Data quality: Some comments on the nasa software defect datasets. *IEEE Trans. Software Eng.*, 39(9):1208–1215, 2013.

[48] Emad Shihab, Christian Bird, and Thomas Zimmermann. The effect of branching strategies on software quality. In *ESEM*, pages 301–310, 2012.

[49] Dag I.K. Sjøberg, Bente Anda, and Audris Mockus. Questioning software maintenance metrics: a comparative case study. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '12, pages 107–110, New York, NY, USA, 2012. ACM.

[50] Dag I.K. Sjoberg, Aiko Yamashita, Bente Anda, Audris Mockus, and Tore Dyba. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 2013.

[51] But he looked good on paper. `http://www.slate.com/articles/business/small_business/2010/08/but_he_looked_good_on_paper.html`.

[52] Sonar. `http://en.wikipedia.org/wiki/SonarQube`.

[53] Margaret-Anne Storey, Leif Singer, Fernando Figueira Filho, Brendan Cleary, and Alexey Zagalsky. The (R)evolutionary Role of Social Media in Software Engineering. In *ICSE*, Hyderabad, India, June 2014. IEEE CS.

[54] Walter F. Tichy. Design, implementation, and evaluation of a revision control system. In *ICSE*, pages 58–67, 1982.

[55] Claude E. Walston and Charles P. Felix. A method of programming measurement and estimation. *IBM Systems Journal*, 16(1):54–73, 1977.

[56] M.R. Wigan and R. Clarke. Big data's big unintended consequences. *Computer*, 46(6):46–53, 2013.

[57] Big data. `http://en.wikipedia.org/wiki/Big_Data`.

[58] Business intelligence. `http://en.wikipedia.org/wiki/Business_intelligence`.

[59] Git. `http://en.wikipedia.org/wiki/Git_%28software%29`.

[60] Predictive analytics. `http://en.wikipedia.org/wiki/Predictive_analytics`.

[61] Vworker. `http://en.wikipedia.org/wiki/VWorker`.

[62] Jialiang Xie, Qimu Zhengand, Minghui Zhou, and Audris Mockus. Product assignment recommender. In *ICSE'14 Demonstrations*, 2014.

[63] Jianming Ye. On measuring and correcting the effects of data mining and model selection. *Journal of the American Statistical Association*, 93(441):120–131, March 1998.

[64] Tze-Jie Yu, Vincent Yun Shen, and Hubert E. Dunsmore. An analysis of several software defect models. *IEEE Trans. Software Eng.*, 14(9):1261–1270, 1988.