The Journal of Systems and Software 101 (2015) 59–68

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Manufacturing execution systems: A vision for managing software development

Martin Naedele [a], Hong-Mei Chen [b], Rick Kazman [b,*], Yuanfang Cai [c], Lu Xiao [c], Carlos V.A. Silva [b]

[a] ABB Network Management, Baden, Switzerland
[b] University of Hawaii, Honolulu, HI, USA
[c] Drexel University, Philadelphia, PA, USA

ARTICLE INFO

Article history:
Received 8 October 2013
Revised 7 November 2014
Accepted 8 November 2014
Available online 29 November 2014

Keywords:
Manufacturing execution system
Software development management
Decision support
Modularity debt management

ABSTRACT

Software development suffers from a lack of predictability with respect to cost, time, and quality. Predictability is one of the major concerns addressed by modern manufacturing execution systems (MESs). A MES does not actually execute the manufacturing (e.g., controlling equipment and producing goods), but rather collects, analyzes, integrates, and presents the data generated in industrial production so that employees have better insights into processes and can react quickly, leading to predictable manufacturing processes. In this paper, we introduce the principles and functional areas of a MES. We then analyze the gaps between MES-vision-driven software development and current practices. These gaps include: (1) lack of a unified data collection infrastructure, (2) lack of integrated people data, (3) lack of common conceptual frameworks driving improvement loops from development data, and (4) lack of support for projection and simulation. Finally, we illustrate the feasibility of leveraging MES principles to manage software development, using a Modularity Debt Management Decision Support System prototype we developed. In this prototype we demonstrate that information integration in MES-vision-driven systems enables new types of analyses, not previously available, for software development decision support. We conclude with suggestions for moving current software development practices closer to the MES vision.

1. Introduction

Despite decades of research into software development processes, software development still suffers from a lack of predictability with regard to the cost, time, features, and quality of delivered products (Naedele et al., 2014; Kaur and Sengupta, 2011; Kanarakus, 2013; Shalal-Esa, 2014; Jones and Bonsignour, 2011). Software development managers often do not have a clear picture of exactly where their teams stand in the delivery schedule, as expressed by the time-honored Ninety-ninety rule: "The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time." (Bentley, 1985). "Today, this challenge is becoming even more complex for software development management, because new issues such as avoiding legal liabilities, global distribution of development efforts, and ensuring employee qualification and retention are being added to the issues to be managed during development. For example, legal liabilities could arise out of violation of open source licenses, copy and paste of copyrighted code from the Internet, or infringement of software patents (Hammouda et al., 2010; Scacchi and Alspaugh, 2012), violation of export control regulations or product defect liability laws.

One solution to better cope with these challenges is to learn from another domain that has comparable planning, monitoring, and tracking needs: industrial manufacturing. In fact, with "components", "product lines", and "software factories", software engineering already has adopted many concepts from manufacturing (Cockburn, 2007). Previous work on software project management has also advocated an "integrated" approach where software processes are to be optimized as an integral, continuous process in the entire life cycle (Abdel-Hamid and Madnick, 1991). Similar to the manufacturing concept, it emphasized the use of the feedback principles of system dynamics to structure and clarify the complex web of dynamically interacting variables that affect management decisions (Abdel-Hamid and Madnick, 1991). This paper takes the idea of learning from manufacturing one step further: to directly adapt the principles of a manufacturing execution system (MES) to software engineering. This, we claim, will help to manage and increase the predictability of software development by more fully integrating information, and will provide a basis for automated tool support for improved decision making.

* Corresponding author at: Department of Information Technology Management, University of Hawaii, 2404 Maile Way, Honolulu, HI 96822, United States. Tel.: +1 412 268 1588; fax: +1 412 268 5758.
E-mail addresses: martin.naedele@ch.abb.com (M. Naedele), hmchen@hawaii.edu (H.-M. Chen), kazman@hawaii.edu (R. Kazman), yfcai@cs.drexel.edu (Y. Cai), lx52@drexel.edu (L. Xiao), carlos.andrade@acm.org (C.V.A. Silva).

http://dx.doi.org/10.1016/j.jss.2014.11.015
0164-1212/

Our intention is not to argue that software development can or should be managed like manufacturing. Our intention is also not to claim that a factory MES product could be or should be used as-is for software development. It is well-understood that, among other differences, software design is associated with higher level of complexity, and the experience and skill of software developers play more significant role in software product quality and productivity. Despite these differences, integrated management of information collection and information flow is equally, or even more important in the software development process given the greater levels of idiosyncrasy and unpredictability in software development as compared with manufacturing. Both software and manufacturing process management requires planning, monitoring, and tracking. Even for non-commercial open source software development, where information on costs, schedule, labors, etc. are not explicitly available, the concern about software quality is no less than commercial software development. As a matter of fact, successful open source projects usually employ strict processes where developers' privileges are tracked and controlled based on information such as the amount of contribution they made to the project (Mockus et al., 2000).

Given that manufacturing practitioners have spent decades refining their planning, monitoring, and tracking methodology, our idea is to learn from the integrated management of information flows in state-of-the-art production facilities from MES evolution in industrial production and customize them to the development of software, instead of reinventing this wheel piece by piece.

Software development as knowledge work is different from many forms of industrial production, and often has many dimensions of unpredictability and non-determinism. Many industrial manufacturing and production processes are characterized by a separation of the creative design and development process and the repeated or continuous production or assembly of the same goods, making use of machines, robots, and unskilled labor. On the other hand, there are manufacturing operations that use highly skilled workers and on the spot creative problem-solving to produce complex small series or unique products based on a set of skills, processes, and semi-finished products. Such production is quite comparable to commercial software development.

For different types of software projects, commercial or open source, the type and amount of information needed may vary, but they could all benefit from the MES-vision-driven development that centers on information integration, which we will make clear in Section 5. Independent of how closely related the actual production operations are in industrial manufacturing and software development, the idea of connecting the various streams of information about people, tools, artifacts, specifications, times, costs, defects, qualifications, customers, etc. is inherently valuable in software development. We will also demonstrate how different types of analysis and decision support functions that were not possible previously can be performed given integrated data in Section 5.

MES principles, which center on information integration, are not new. However, their full usage still remains a vision for software development. We have observed, in our industrial practice, that the lack of integrated information hinders our planning of software development and prevents easy access to information required for management decisions (Naedele et al., 2014). In Naedele et al. (2014) we presented five scenarios that motivate how different concerns of software development management could be supported by a MES-vision-driven system. In this paper we describe in detail how such a system could be architected, detail the gaps between existing software development practices and the MES vision, and provide some early results that show that such a MES-vision-driven software development is both feasible and beneficial. The results of this paper should be useful to both software development organizations and software engineering tool vendors. Software development organizations can identify their tracking and tracing gaps, as compared with best practices in manufacturing, and address these gaps with commercial or in-house tools. Tool vendors can take inspiration from this paper on how their integrated development environments or application lifecycle management tools could be extended.

In what follows, we introduce basic MES concepts. Section 3 details MES functional areas applicable to software engineering processes. Section 4 describes the gaps between MES-vision and existing software development practice. In Section 5, we present a prototype system called the Modularity Debt Management Decision Support System (MDM-DSS) that demonstrates the feasibility of how some of the gaps identified above can be closed and how a MES-vision-driven system could be architected. Section 6 provides four examples of new analysis and decision support functions in MDM-DSS that are enabled by the MES vision. We discuss the challenges in implementing MES-vision-driven systems in Section 7. Conclusions are provided in Section 8.

2. Manufacturing execution systems

Manufacturing execution systems were borne out of the need to deal with progress, resource, maintenance, quality, and provenance monitoring and tracking requirements in manufacturing in the 1980s (MESA, 1997; ZVEI, 2011). A MES is essentially a decision support system for managers.

The promises of MES include better planning and estimation accuracy, support of process improvement through metrics, better staff and talent management, lower production costs, better management of compliance as required in aerospace, food and drug industries, better supplier management, shorter time to market, alignment of local production objectives with global enterprise goals, and reduction of manual data entry (for time recording and billing purposes).

To realize these promises, MES functionality covers the data collection and analysis of three aspects of manufacturing:

- Past: Storage and analysis of historical data of past process runs.
- Present: Real-time open and closed loop control of the process.
- Future: Prognostics and planning of future process runs as well as early warning of process or quality deviations.

There are a few reference models created by organizations such as the Manufacturing Enterprise Solutions Association (MESA), ISA, NAMUR, and the German VDI to define largely overlapping feature sets for MES. As discussed in Naedele et al. (2014), the MESA reference model (http://www.mesa.org/en/modelstrategicinitiatives/MESAModel.asp; MESA, 1997), which has been substantially revised since its inception in 1997, describes 10 functional areas and data dependencies among them. These 10 areas are (1) labor management, (2) resource allocation/status, (3) dispatching, (4) product tracking, (5) quality management, (6) performance analysis, (7) process management, (8) scheduling, (9) document control and (10) maintenance management.

A detailed explanation of the scope and content of each functional area can be found in ISA 95, part 3 (Standard ANSI/ISA, 2007); there exist overlaps and interrelations between the functional areas. Implementation varies, though. Also, between different plants, the emphasis on the different MES functions will vary, depending on the industry and the specific production and profitability bottlenecks the plant faces. Some aspects may even be missing in a given MES implementation. Similarly, we do not require or expect that MES-vision-driven software development will include these 10 functional areas entirely and address their data sources to the same degree. Any implementation will be driven by individual organization's unique business context, for instance, what the most urgent supervision and decision goals for the organization are. We focus on the principles of MES and the idea that all management tools should exchange information and make use of that information to do their jobs better, not on the completeness of the functional areas.

3. MES features applied to software engineering

In this section, we explain the specific functional areas of a MES and how they could be applicable to improve the software development process and address software development challenges/concerns. The first and foremost principle of MES is that these 10 functional areas are all based on product processes in common use, and the data that may be collected from the processes. In manufacturing MESs, raw data are constantly collected, recorded, and organized from production processes for use in the other areas. Collection of measurements is the most important factor for achieving a controlled process. Although this is also true in theory for software development, in practice, automated data collection and acquisition in software development projects today are not common (Montaser, 2013).

In software projects, a wide range of measurements are needed, including not only software artifact data such as function points, number of bugs, lines of code, number of test cases, etc., but also quality and productivity data related to developers, such as the number of lines of code committed per unit time per person or the number of defects resolved per developer or team. However, if data are collected on the level of individual developers, controls have to be put in place to ensure that these data are not available for individual performance reviews; otherwise the goal of process improvement would not be achievable (Austin, 1996).

Based on the data collected from the various processes, a manufacturing MES supports the following 10 functional areas. We discuss below how each functional area can be applied to software development processes and the associated challenges.

3.1. Labor management

Labor management directs and tracks the availability and use of production staff based on skills and qualifications as well as constraints such as absences. Managing staff is an important concern in software development as well. In fact, these concerns are more prominent with globally distributed software, which adds another level of complexity—the lack of face-to-face interaction—and thus requires more disciplined labor/staff management. Comprehensive information about developers, beyond those needed in traditional MES systems, could be taken into consideration during software development task assignment.

3.2. Resource allocation and status

Resource allocation in MES is about directing and tracking the specific mapping of people, tools, and materials to tasks. In software development, the major resources that need to be allocated are people. For new development, especially when using agile practices, task allocations are typically informal and driven by availability and personal interest of the developer selecting the task, but not by organizational aspects like cross-training needs. Other scarce resources might need to be managed as well, e.g. licenses for special tools and access to prototyping boards and hardware debugging tools. These resources are typically managed in informal and ad-hoc ways today.

3.3. Dispatching production units

While the previous resource allocation step deals with matching tasks with suitable developers, the dispatching step deals with the decision of which tasks to work on next, given priorities, dependencies, and resources. Similar to resource allocation, dispatching of software development tasks is typically done by team leads in informal ways based on lists of requirements and schedules captured in project management plans. Dispatching of test tasks may be more formal, if test plans and test management are used. Build and integration tests are often automated and triggered by code check-in into the software configuration management system.

3.4. Product tracking and genealogy

A mass-market software producing organization might not be interested where their products are sold to, installed, and used, and thus may have no need for product tracking. However, in an increasing number of application domains, the software business is not a one-time sales event; the initial sale is just the starting point for a continuing series of vendor-initiated updates. For that purpose, the software vendor needs to know which customer has which versions of the software so that communication to customers and update deployments can be planned and executed in an efficient way. Another driver for the need to know exactly which version and configuration is installed at what site arises whenever the software producer is taking responsibility for supporting its customers in removing security and safety related vulnerabilities or defects discovered after release, like with the recent "Heartbleed" vulnerability in SSL libraries and products using them. Today there is an increasing expectation that software vendors accept such responsibility, and at some point this might even become legally mandated in some industries.

Tracking provenance of contributions to the software product is important for three reasons: The first one arises from intellectual property and copyright law. It is necessary to ensure and document that no unlicensed IP is used in the developed application. The second reason results from other legal requirements that require proving that developers involved in the creation of certain artifacts have certain qualifications (e.g. Professional Engineer's license, safety engineer certification, etc.) or meet other personal constraints like citizenship or security clearance. A combination of product genealogy and labor management functions can achieve this. Even if not subject to requirements on the level of individuals, export-control regulations of certain countries require that a vendor is able to document the geographic regions from which certain subsystems—for example, cryptography—originate. In a more general case, there are various standards that require the collection and presentation of evidence to certifying agencies that certain software and system engineering process requirements were followed. This compliance documentation for certification according to e.g. IEC 61508 (safety) or IEC 15408 (security Common Criteria) is effort-intensive and expensive to create after the fact or without integrated tools. Concerns about backdoors or other intentional security vulnerabilities or malicious functionality give a third motivation for provenance tracking. A customer may want to know about the origin of all the pieces in a software application or embedded system to assess the risk of malware infiltration.

To deal with the challenges described above, a software MES application should be able to search the produced code base for collisions with known problematic source code and also to securely manage detailed certificates of provenance/originality provided by developers together with each piece and version of code they submit to the software configuration management system. For each product and version the MES should be able to provide a software "bill of materials" annotated with tamper-proof provenance information.

3.5. Quality management

In a MES, quality management involves recording and tracking measured quality parameters of work products and processes, comparing them against targets, and triggering reactions on below-target quality. Many software development tools have APIs for metrics calculation, but such data collection rarely happens in an integrated fashion and with broad coverage. In a software project, any one of the metrics alone is usually not sufficient to provide a basis for management decision making or pinpointing problematic processes or artifacts. Given the wide acceptance of issue tracking and version

control systems, it becomes possible to trace each defect to concrete times, versions, and developers. Although a large number of research papers have proposed leveraging such data for the purpose of various analyses, such as defect prediction, an integrated and systematic data collection and synthesis mechanism is not available today.

3.6. Performance analysis

In a MES, performance analysis includes measuring task execution parameters, calculating key performance indicators (KPIs) e.g. on quality, availability, productivity, and comparing them to targets set by the organization or external regulatory bodies, as well as presenting and visualizing these KPIs for various stakeholders e.g. upper management, human resources, production management, testers/QA, product management, sales, supply management, process group, maintenance, logistics.

In software development projects, performance analysis for different stakeholders is similar. A variety of process and product related metrics is available for that purpose (Jones, 2008). Reaching and maintaining CMMI levels 4 and 5 requires measurement-based performance analysis. While some performance analysis driven feedback loops are obvious, one could also imagine more sophisticated applications, for example to give software architects early feedback on whether the proposed architecture is being accurately or efficiently realized by an organizational setup in a distributed team. As another example, planned and actual schedule could be tied back to requirements and their valuation to enable real-time earned value control for projects.

Care has to be taken to manage the trade-off between detailed analysis of quality and performance data and incentive-relevant performance evaluation for individual employees and units (Coman et al., 2009) which might produce unintended and counterproductive side effects (Austin, 1996) or even result in conflicts with local labor laws and trade union requirements. A software MES could ensure that only suitably aggregated information is available to each stakeholder.

3.7. Process (development progress) management

Process management in MES involves directing and tracking the flow of work through the plant, creating alarms in case of deviations and providing decision support to correct deviations or react on other events, including approval workflows and escalation management. Tools to detect and provide guidance on process deviations in software development are available, for example, in the form of build systems that send out alerts in case of broken builds, smoke-test type integration tests, regression test suites, as well as software visualization (Maletic et al., 2011) and impact analysis tools to support the development team in finding the least risky approach to making necessary changes.

3.8. Operations/detailed scheduling

In a MES, scheduling involves optimal sequencing of tasks considering finite resource capacities and other constraints. Support for scheduling in software development exists in the form of online and offline requirements prioritization tools, traditional project management scheduling tools, and backlog management tools.

3.9. Document control

In a MES, document control involves distributing relevant information to the people working on tasks at the right time (e.g. process documentation, design documentation, work orders), and collecting new documents resulting from production (e.g. design documentation, test/QA documents, certifications of provenance). Industrial manufacturing process engineering expends much effort on making clear and concise instructions about the task at hand and making necessary supporting information available to the worker at the right time and place. The goal is to minimize the time the worker has to search for information or tools. Part of the document control functionality of the software MES could thus also be interfaces to experience management systems (Basili et al., 1994), both as recommender system to provide useful experiences to the developer just in time based on project context, and to feedback usage and benefit information into the experience repository.

In software development, there is typically a document management system for long-term storage and version management of documents. Beyond that, software developers experience little support from systems that provide selected documentation for the task at hand. Information like related requirements, design descriptions, and issue tracking tickets are usually kept in separate systems and must be manually integrated by the developer.

3.10. Maintenance management

In a MES, maintenance management involves collecting statistics on tool performance and uptime, and planning improvement work and investments. As there are few physical tools subject to wear involved in software development, machine maintenance is of low importance. Nevertheless, it is clear that software (code) changes much more frequently than hardware and software maintenance—in the sense of accommodating new features—can consume up to 80% of overall project effort.

In addition, one could imagine that slowdown and downtimes of the IT infrastructure for software or hardware upgrades or virus scans could be integrated with the scheduling and dispatching tools to avoid unnecessary waiting times. Other maintenance aspects related to tools usage and user satisfaction might trigger tool replacement or user training. Time series data on the usage of tools can be used to identify where time is spent (and, more important, lost) in the project and thus find which process elements are inefficient and cause bottlenecks.

4. Gaps between existing software development tooling and the MES vision

The previous section has shown that there exists considerable potential for productivity, efficiency, and quality improvements in lending manufacturing-proven MES functionality to the software development domain. However the mapping of MES functional areas to the software development process is, not surprisingly, imperfect. Employing our understanding of the unique properties of software and its development, we now analyze the major gaps between state-of-the-art software engineering and MES practices. Note that we are not claiming that these gaps are impossible to bridge—quite the contrary. There are many challenges. However, we do claim that existing software engineering toolsets today largely ignore these gaps, to the detriment of the systems and projects that these toolsets purport to support.

4.1. Gap 1. Lack of integrated use of people data

Labor management in software development is today typically addressed by standalone project management and time accounting tools like MS project. Time accounting is usually coarse-grained and can rarely be traced back to specific development tasks. Qualification and talent management and personal or career development aspects are seldom incorporated into such tools and their related processes. Visibility into, and resource planning of, team members is rarely achieved and this is even more difficult when the team is globally distributed. Time series data could also be used to monitor and evaluate the success rates of staff training.

Compared with other types of software project data, such as evolution history (recorded by version control and issue tracking systems), and software artifact data (such as source code measurements), data about people are typically not recorded as part of the software project management process. And such data, which are commonly available in HR systems or ERP (enterprise resource planning) systems, is seldom used for the management of software development teams.

For example, consider what happens at a company if a project within the company wants to free some time of architect A (so that A can work on other things). It is not clear what tasks A does, how much time is spent on each, which of this work is non-architectural and could be done by other team members, and which team members would be best suited to take up these tasks from a loading, qualification and career development perspective. Currently most software development organizations lack the integrated data needed to support such decision-making.

4.2. Gap 2. Lack of effective, unified data collection infrastructure

Although some of the data needed in software development MES exist in various tools in current use, these data have heterogeneous formats associated with their own tools for their own purposes. Revision history and bug tracking data, for example, are usually managed by separate tools, even though they are closely related. Requirement and design documents may be created, but their evolution is not as well managed and controlled as that of source code and the management of these documents is disconnected from other artifacts, such as code and test cases. More seriously, in large organizations, architecture design, quality control, and requirements engineering are usually managed by different departments, using separate tool suites, making it harder to collect data uniformly.

Some research work has been done on building infrastructures for software engineering data collection (e.g. PROM (Coman et al., 2009), Hackystat (Johnson, 2007), CodeMine (Czerwonka et al., 2013)) but despite case studies reporting positive results, they only provide part of the envisioned MES for software, and in any case have not been widely adopted. To enable meaningful fine-grained analysis or even closed-loop control of development processes, it will be necessary to collect time-series data sampled at frequent regular intervals instead of collecting only data associated with events such as code check-ins.

4.3. Gap 3. Lack of common conceptual frameworks driving improvement loops from development data

To achieve a MES-vision-driven software development process, it is necessary to combine and correlate measurements and data from different areas. For example, accurate size and effort estimates depend on collecting high-quality historical data about the effort needed for implementation tasks together with related parameters such as the qualification levels of developers and achieved code quality or defect levels (Buglione and Ebert, 2007). The degradation of productivity and quality are usually the result of architectural decay. Thus analysis methods to close the loop between architecture and implementation are necessary.

To build analysis, prediction, and warning systems, we must have commonly agreed on metrics and indicators, standardized APIs for data extraction from tools, and benchmark baselines valid for different types of development projects, e.g. open source or commercial applications.

4.4. Gap 4. Lack of support for integrated future projection and simulation

In manufacturing MES, scheduling involves selecting the tasks to accomplish in the near future under constraints such as limited resources, and aiming to optimize the outcomes. One important element in scheduling is projection of the near future: which requirement items should be addressed in the next release? Which bugs should be fixed first? Which parts of the code should be refactored? What will be the consequences of refactoring, or not refactoring, in terms of costs and benefits? Unfortunately, while there is some methodological support for combining architecture analysis and cost/benefit analysis (Nord et al., 2003), there are no tools or frameworks that allow for the future projection and simulation that are necessary for effective scheduling and dispatching.

In summary, given the recent advances in software engineering management tools, such as version control systems, issue tracking systems, backlog management systems, as well as social media that support collaboration and communications among developers, most of the data needed as the foundation for a software MES are readily available. However, due to the fact that these data are produced from separate tools, and usually managed by different departments of a large organization, the framework needed for information integration and acquisition are not yet available. Moreover, the information about people, which is usually managed by human resources, should be integrated with other project data generated from the development process. Finally, a well designed, MES-driven decision support system for software development should support future projection and simulation.

5. The MDM-DSS

In this section, we present a prototype system we have been evolving for the past 3 years called the Modularity Debt Management Decision Support System (Cai et al., 2014) that demonstrates the feasibility of how the gaps identified above can be closed, moving us toward the vision of MES-driven software development. The MDM-DSS is illustrated in Fig. 1.

The goal motivating this example is *software quality management*. Specifically we are attempting to measure and manage modularity (design) debt in complex software projects. To achieve this we have applied the MES principles and built a *data collection infrastructure* (addressing Gap 2) that forms the foundation for a decision support system. The inputs to the system, including source code, patches, bug reports, commits, etc., are the outputs of a typical software development process. The ETL (extract, transform, and load) modules, text mining tools, as well as the project data warehouse, is similar to the data collection and acquisition infrastructure of a MES system. The modularity debt metrics, effort prediction models, and simulation models are equivalent to the components of specific functional areas of a MES, such as quality management and performance analysis. The modularity debt prediction, hotspot identification, refactoring cost/benefit estimation, and refactoring simulation can similarly to be seen to be equivalent to the MES areas of resource allocation, maintenance management, and scheduling.

In the MDM-DSS infrastructure we collect data from projects and people (addressing Gap 1)—data on the source files, the changes to those files over time (patches and commits), the bugs and issue reports affecting the system, and their associated discussions—and store this data in a project-specific data warehouse. We model people data as well as mining associated discussions to estimate effort. So, for example, we can determine profiles of different programmers with respect to how effective they are at removing bugs, their level of expertise with different areas of the project's code-base, their productivity, their experience with related frameworks and tools, and so forth (Kersten and Murphy, 2006).

Note that in the MDM-DSS we can also make use of data *external* to a given project. This is particularly useful if a project is young and hence does not have a long history on which to base decisions. Using external project data allows decision-making to proceed based on a set of company-wide or industry-wide standard projects.

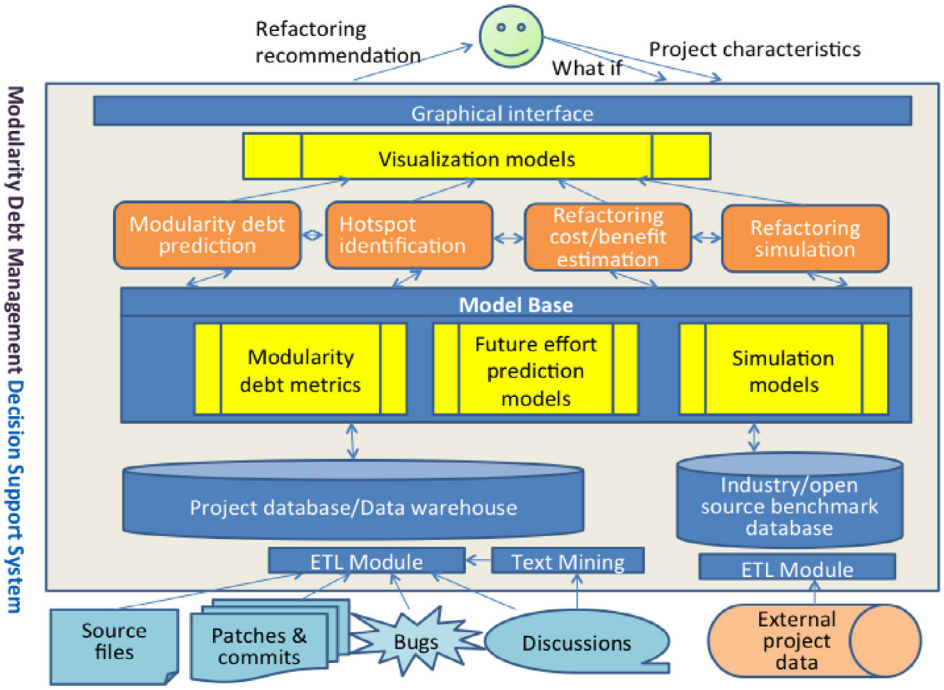

Fig. 1. The Modularity Debt Management Decision-Support System (MDM-DSS).

To move toward narrowing Gap 3, we extract software complexity metrics and project effort measures. Using these measures we can determine areas of potentially high modularity debt. But just because a piece of code is complex does not mean that we want to spend project resources to pay down the debt. By mining project history, we can determine if the identified areas of the code base are also sites of anticipated future change. After all, we do not want to spend precious project resources to "fix" a part of the code that seldom changes. For this reason we need to simulate the future, to determine the potential payoff associated with reducing technical debt (Brown et al., 2010) (for example, by refactoring). All of this information can then be presented to a manager, as a set of recommendations and the manager, possibly playing out a number of different what-if scenarios, can make informed decisions based on the anticipated costs and potential benefits of various refactorings. This is an example of showing how Gap 4 can be narrowed.

More specifically, to address Gap 4 (lack of support for integrated future projection and simulation), we are experimenting with the use of the Datar–Mathews method for valuing real options (Mathews and Datar, 2007) as a means of valuing re-modularization (refactoring) activities. A real option is the right, but not the obligation, to make an investment in the future. The Datar–Mathews method was originally developed for use in the Boeing Corporation and is more transparent and intuitive for managers to apply while being analytically equivalent to Black–Scholes. Furthermore, it can be easily implemented using Monte Carlo simulation. Monte Carlo simulation typically consists of a mathematical framework and an accompanying tool suite. This allows us to simulate the value of proposed modularization activities, under scenarios that we choose. The value of refactoring a specific part of a software corpus, for example, is tied to how that part is expected to evolve, which is in turn dependent on user requirements. The real option value of a refactoring activity is calculated by simulating the following simple formula using a simulation package, such as Oracle's Crystal Ball or Palisade's @Risk:

$$\text{Real option value} = \text{Average}[\text{MAX}(profit - cost, 0)]$$

where *profit* is a distribution of the discounted profits at time 0 and *cost* is the estimated cost of implementing the specific re-modularization activities.

What we are trying to model with this method is straightforward: the more often you expect to modify a module, the more value you will gain by "purchasing the option" to make that module easier to modify and maintain. For example, if the business logic in an e-commerce site is frequently changed, then there is benefit in having this logic be "easy to change" (Carriere et al., 2010). To estimate the benefit of refactoring the business logic, stakeholders need to estimate the future volatility of that portion of the code: *how often is business logic expected to change and in what ways?* A project manager or architect can make these estimations based on intuition or, better still, on project history derived from the data. That is, the modules that are changed most frequently and most recently will be automatically proposed by the MDM-DSS as candidates. The triangular distribution used in the simulation simply requires that the architect (or other project stakeholder) estimates an optimistic, pessimistic, and most likely value for any estimated parameter (such as the number of expected times that this module will be modified in a given time period). Given the expected *number* of changes to the module, we simply multiply it by the *expected costs savings* (due to the proposed modularization activity) per change to determine the total benefits. The architect can leverage the data collected from project history in the MDM-DSS to estimate these cost savings—the benefit—and can use the module identified as a refactoring candidate to estimate the cost of refactoring.

Given the benefit that accrues to this change along with the anticipated frequency of the change, we can now estimate the *net* modifiability benefits of the change at any time period in the future. An example of how this calculation would be formulated is given in the spreadsheet shown in Fig. 2, where the Monte Carlo simulations are performed by the @Risk software add-in. In this example project, the current code structure complexity increase factor is 9% and a proposed refactoring promises to bring the number back to 5% (Cai et al., 2014).

6. Instantiating the MDM-DSS: four examples

Over the past 3 years we have used this infrastructure to collect data from, and to analyze, more than 20 open source projects and 1 proprietary project. For example, in Cai et al. (2014) we previously

Project Risk Rate	Investment Rate	Inflation Rate	Code Complexity Adjustment	Ref'd Code Complexity Adjustment	Code Defect Rate	Refactored Code Defect Rate	Average Cost/ Defect	Average Refactored Cost/Defect	Code Size (KLOC)
15%	5%	3%	9%	5%	0.003	0.0026	$ 320	$ 250	52000

Year	Operating Profit NPV	Refactoring Cost NPV	Expected	Cost/Change with Refactoring			Expected	Cost/Change without Refactoring			Defect Reduction Benefit
				Pessimistic	Most Likely	Optimistic		Pessimistic	Most Likely	Optimistic	
2010	$ -	$ 42,833									
2011	$ 9,412	$ -	$ 841	$ 1,300	$ 900	$ 700	$ 1,449	$ 2,300	$ 1,500	$ 1,200	5178
2012	$ 11,651	$ -	$ 811	$ 1,460	$ 973	$ 786	$ 1,415	$ 2,582	$ 1,684	$ 1,347	6022
2013	$ 13,880	$ -	$ 783	$ 1,639	$ 1,053	$ 882	$ 1,381	$ 2,899	$ 1,891	$ 1,513	6933
2014	$ 14,780	$ -	$ 756	$ 1,840	$ 1,138	$ 991	$ 1,348	$ 3,255	$ 2,123	$ 1,698	7444
Total	$ 49,723	$ 42,833									
ROV	6890										

# Changes/ Year			# Defects Year			Refactoring Cost		
Pessimistic	Most Likely	Optimistic	Pessimistic	Most Likely	Optimistic	Pessimistic	Most Likely	Optimistic
12	7	5	180	156	130	$ 51,000	$ 42,500	$ 35,000
18	11	8	210	182	150			
28	14	11	240	204	180			
30	21	14	260	220	190			

Fig. 2. An example of a spreadsheet realizing the Datar–Mathews model (Cai et al., 2014).

reported on some of the results of analyzing five Apache projects: Derby, Lucene, PDFBox, Ivy, and FtpServer.

The software metrics that we have collected thus far are primarily coupling and cohesion-based, and focus on capturing structural/architectural complexity. We have chosen to use the Chidamber and Kemerer suite of metrics (Chidamber and Kemerer, 1994), as these are well-established and validated, as well as a more recent coupling metric, propagation cost (MacCormack et al., 2006). But in principle any appropriate metrics could be employed in an MES.

In addition to these metrics, we collected proxy measures of effort, on a file-by-file basis, so that we can analyze the amount of effort that goes into fixing a bug or adding a feature. This aids us in drawing conclusions about the use of personnel. We collect *proxy* measures because actual measures of effort are seldom available in software projects and even when effort has been tracked, it is typically tracked at such a coarse level of granularity (i.e., the project, or major subsystem) as to be of no use in supporting analysis, The proxy measures of effort that we collect are: *actions*—the total number of patches and commits made to a file for the purpose of addressing issues, *churn*—the number of lines of code changed in a file for the purpose of addressing issues, *and discussions*—the number of textual comments made to address issues and related to a file. Furthermore we capture the number and severity of bugs that affect each file. Finally, we employ a modularity violation detection tool to identify trouble spots in the code base (Schwanke et al., 2013).

Using these multiple metrics, and measures, we are able to support a manager in "triangulating" problems affecting the (files of the) software project, as we will show. But just identifying potential trouble spots is not enough. The purpose of a MES-vision-driven software development process, as stated above, is to give management better visibility into processes and to enable them to react quickly and appropriately. Reacting quickly and appropriately means balancing multiple factors, including the future quality of the product, impending requirements and associated deadlines, the available skills of the software developers, and the costs of development, maintenance, and operations. In the end, virtually every decision in a software project is an economic decision.

For example, we can calculate a "maintainability risk" measure for each file in a software project using existing software metrics. A manager can monitor the *variation* of these measures over the course of the system's evolution, using the system to rank the modules based on their maintainability risks. The manager can also monitor the variation in the number of bugs, the time needed to fix the most critical bugs, the time spent by each developer on each task, etc. The data collected using this MDM-DSS can be used to support software-related decision making, including defect and effort prediction and refactoring decisions.

To achieve this goal we have created a tool suite to extract data from various open- and closed-source projects. These tools analyze the code base itself, the revision history, and the log of bug/issue reports and their associated discussions, thus addressing Gap 2 (unified data collection infrastructure).

Example 1: Fig. 3 depicts one example of time series data that we have collected—the number of change requests—for four open-source projects. The data is collected in 2-week intervals, a typical time interval of a sprint for an agile project. This graph shows how the "activeness" of a project, as measured by the number of change requests, varies over time and how the activeness differs between projects. This is a global measure of project health. For example, the graph shows that the activeness of Lucene increases over time whereas the Xerces project has far fewer change requests, as compared with other projects. The number of actions can be as an important KPI of the project, which is within the performance analysis area of a MES.

This is the simplest "software MES" function that can already be easily collected and visualized in most open source and commercial projects, and thus just addresses Gap 2. In our prior work (Schwanke et al., 2013), we have also collected data about the types of tickets recorded in a commercial project's issue tracking system over

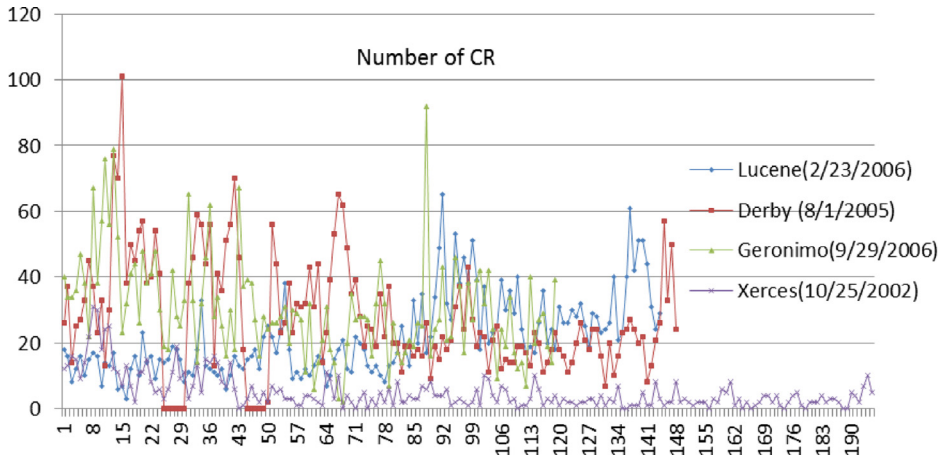

Fig. 3. Change requests in Lucene, Derby, Geronimo, and Xerces.

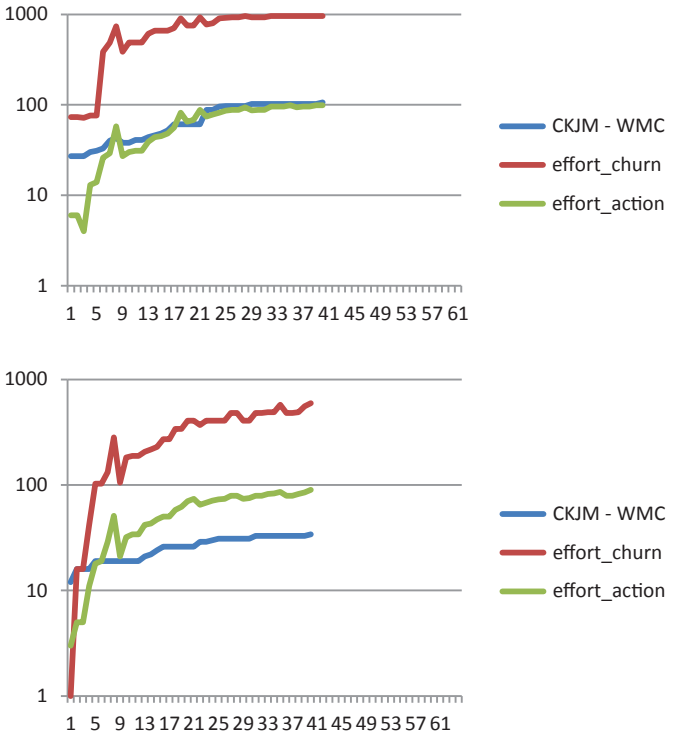

Fig. 4. Effort and complexity tracking in the Apache Camel project.

multiple releases. The data clearly showed that, as time goes by, more and more effort was spent in fixing bugs, rather than adding new features, revealing a declining feature delivery velocity. This is a simple form of addressing Gap 4.

Example 2: Let us consider another example. This data, collected from the Apache Camel project, analyzed the riskiest files in the project, as measured by the number of bugs that have historically affected them. For each of these files we calculated complexity metrics and measures of effort. Fig. 4 below shows the progression of one such metric—weighted method complexity—and two proxy measures of effort—churn and actions, for two of the project's files. These values have been plotted over 61 versions of the files CxfEndpoint.java and BeanInfo.java, where a version corresponds to an agile sprint typically lasting 2–4 weeks. In each of these graphs, the *X*-axis is the version number and the *Y*-axis is the value of the various metric and effort proxy measures, as plotted on a log-10 scale. As can be plainly seen, as the complexity of each file goes up, the effort to evolve, fix, and maintain the file increases dramatically. Because these measures can be automatically tracked, prioritized, and visualized for a manager, "hot spots" in the code base can be easily identified and analyzed, thus addressing Gap 3 and Gap 4.

This information belongs to the maintenance management area of the MES. Such information provides continuous feedback to the manager, supporting a principled data-driven analysis of how to allocate project resources, in the spirit of a MES. The data used in this example were collected and analyzed using the same MDM-DSS system as Example 1. The more data are collected and integrated, the more MES areas can be applied.

Example 3: We recently (Schwanke et al., 2013) conducted a case study using a real industrial project. In this study, we first identified "hotspots" of the system by combining source code metrics with history data extracted from the project's version control and issue tracking systems. We found that most error-prone files are also outliers in terms of files metrics, such as file size and fan-out. During this process, we also realize that only pointing out *which* files are error-prone and too big are not really helpful to the developers. They need concrete suggestions about *how* to refactor.

To further analyze the reasons behind the error-proneness of these files, we analyzed modularity violations that happened during the evolution history of the project. The results show that most files that are changed together but that are not syntactically related, are part of an architecture violation. For example, there was one group of seven files that were put in the incorrect layer, which led to them to never be properly tested. Other reasons behind modularity violations include un-encapsulated assumptions, implementation errors, etc. Based on these results, the project lead compiled a detailed refactoring proposal, which was approved by management and recently implemented. This was another instance of addressing Gaps 3 and 4—our integrated data collection and analysis infrastructure allowed us to drive improvement loops supporting optimized future allocation of project effort.

This case illustrated that the information collected and analyzed can influence the scheduling function area of the MES-vision-driven system for software development. The fact that the refactoring decisions had, in the past, been made after a labor-intensive process with little tool support indicates the need for an integrated framework so that such decisions can be made in a more effectively and timely fashion, and hopefully before significant amounts of maintenance cost have been paid.

Example 4: In another study we explored the relations among the buggy files of 1 industrial and 10 open source projects, expressed

using design rule spaces (DRSpaces)—a group of architecturally connected files, a subset of all the files in the system. The files within a DRSpace are directly or indirectly related to one or more leading files, through one of more types of relations, e.g. inheritance, dependency, aggregation, etc. (Xiao et al., 2014). Leading files are typically important abstraction interfaces, and so many other files depend on them, directly or indirectly.

In our study of DRSpaces we showed that despite the varying sizes and lengths of evolution history among these 11 projects, the number of DRSpaces needed to capture the most buggy files remained stable over time, meaning that buggy files are typically architecturally connected. Also, the more buggy the files, the more closely they are connected within just a few DRSpaces—80% of the top 10 percentile of the most buggy files are captured by no more than 4 DRSpaces, meaning that buggy files are not islands; they influence each other. We also looked at the evolution of the DRSpaces that contain the most buggy files, which we call the "buggy roots". We found that these roots grow with project size and the number of buggy files, meaning that as software evolves more files are connected by these roots, and hence these files become buggy too. Our results indicate that architectural relationships have significant contribution to the bugginess of files and overall project quality, and thus it is critical for a project manager to pay attention to and devote resources to architectural quality as a key to reducing the bugginess of a project.

This case showed that combining information from disparate data sources—bugs from the issue-tracking database, code history from the revision control system, and the code itself—we could gain an important insight into the root causes of project bugginess that could not have been gleaned from looking at any of the information sources independently. This insight can then be used to improve the project, thus addressing Gap 3.

7. Discussion

Our experiences of building this infrastructure revealed several challenges of developing MES-vision-driven system for software development. First of all, different tools provide different types of APIs and interfaces. As a result, although our current framework can uniformly process all the projects using the JIRA issue-tracking system, combined with the SVN version control system, we have to create a new set of tools or substantially modify and parameterize existing tools for projects using different kinds of version control and bug tracking systems. Existing tools simply lack standardized interfaces and protocols for software project data extraction. Second, we similarly do not have standardized interfaces for metrics calculation. We have to revise our tools to accommodate different metrics calculation tools.

We believe that the solution to these problems—increased standardization and interoperability among tools—would help the software community move rapidly toward a MES-vision-driven system for software development. In addition, there is a risk that a software project manager might be a victim of "information overload". The MES principles can also help here, in much the same way that data warehouses help in validating, cleansing, processing, and summarizing business data so that effective decisions can be made. Data, by itself, can be overwhelming, but properly summarized, analyzed and presented, it can be a huge competitive advantage.

Hopefully it is clear from the above examples that not all 10 MES areas apply to all projects, and our approach to add value as long as the MES principles (integrated views of data) are applied even with limited scope of data. In a closed source environment one might have access to more and more kinds of data (e.g. personnel data), but one can still get value out of analyzing any project, even with a subset of the 10 MES areas, as we have demonstrated.

While we have shown that this approach is useful for both open and closed source projects, in the closed source world you have more opportunities to have more kinds of data sources (e.g. HR data, cost data) and more data sources will facilitate more kinds of analysis. But we still believe that this has value in the open source context. After all, nearly all of our examples come from open source systems, and we were able to analyze the publicly available data, combine it, learn from it, and identify obvious areas for improvement.

8. Conclusions and future work

In this paper, we have analyzed how MES principles and concepts can be leveraged to address the concerns of software development in practice. Our contributions are two-fold: first, we analyze and custom-fit the MES-vision to provide software development practitioners a framework (planning, monitoring and tracking) for increasing software development predictivity and quality. Some MES functionality is already in place today in many companies, such as progress and quality dashboards, metrics collection, and time accounting, but they are typically spread over a patchwork of non-integrated systems and tools. In addition, many of the functional capabilities of MES are not being supported in current software project supporting tools. We have identified four gaps between current software system development practice and the MES vision for software development.

Secondly, we have developed a prototype, MDM-DSS that demonstrates new types of analyses (which were not possible, or difficult to perform, previously) enabled by the MES-vision for software development decision-making. The individual capability of each MDM-DSS module is in fact expanded or enhanced by the integration of all functional areas—which is the key point discussed in our examples, illustrating how the four gaps are narrowed in our MDM-DSS prototype. The feasibility demonstrated through our prototype MDM-DSS also sheds light into the challenges and future directions of implementing MES-vision-driven systems. In fact, similar to what historically drove MES deployment in manufacturing, software development organizations may not be able to meet some near-term demands on quality and legal compliance without integrated information management and integrated tool support. We believe that this vision will, of necessity, become a reality in the near future; the only question is how soon and how well we take on this challenge as a community.

Applying MES principles, we have also demonstrated via the MDM-DSS prototype, the feasibility and benefits of using a unified infrastructure to automatically collect, fuse, and analyze data from multiple tools and across different software projects. In our future work, we plan to expand our prototype to use historical effort, change, and bug data to track and infer the costs of code changes and bug-fixing. We intend to extend our case studies of MES-vision-driven software development in different contexts of project management. The goal is to be able to monitor project progress, control quality and make timely project decisions in software development on a sound quantitative and economic basis, just as MESs do for manufacturing systems.

Specifically and as an example, we are already building on our recent work in automatically detecting architectural hotspots (Xiao et al., 2014). We can include these identified hotspots, along with an estimate of the technical debt that they are incurring, in an architect's dashboard—an instance of the MDM-DSS as presented in Fig. 1. We can not only identify the architectural hotspots, but we can provide evidence-driven explanations to the architect as to why these are flaws, which can lead them to understand how to fix the flaws through refactoring. Finally we can provide future effort predictions, based on project data, as to the benefit of a proposed refactoring, allowing architects and project managers to make reasoned economic decisions about software structure. We have already completed one case study realizing this vision and are actively working on others.

We expect our framework based on the MES vision will aid decision support in many other areas of software development. And each of these is a potential area of future research—e.g., planning, tracking

and monitoring of globally distributed development efforts, resource allocation and dispatching of software development tasks, legal compliance of software licensing, tracking of software provenance, software defect prediction, dynamic development effort progress management, real-time earned value control for projects, and so on.

References

Abdel-Hamid, T., Madnick, S., 1991. Software Project Dynamics: An Integrated Approach. Prentice-Hall.

Austin, R., 1996. Measuring and Managing Performance in Organizations. Dorset House.

Basili, V., Caldiera, G., Rombach, H.D., 1994. The Experience factory, Encyclopedia of Software Engineering. John Wiley & Sons, Inc., pp. 469–476.

Bentley, J., 1985. Programming pearls. Commun. ACM 28 (9), 896–901.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N., 2010, Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, pp. 47–52.

Buglione, L., Ebert, C., 2011. Estimation tools and techniques. IEEE Softw. 28 (3), 91–94.

Cai, Y., Kazman, R., Silva, C.A., Xiao, L., Chen, H.-M., 2014. A decision-support system approach to economics-driven modularity evaluation. In: Economics-Driven Software Architecture. Elsevier.

Carriere, J., Kazman, R., Ozkaya, I., 2010. A cost-benefit framework for making architectural decisions in a business context. In: Proceedings of 32nd International Conference on Software Engineering (ICSE 32), Capetown, South Africa.

Chidamber, S., Kemerer, C., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20 (6), 476–493.

Cockburn, A., 2007. What engineering has in common with manufacturing, and why it matters. Crosstalk 20 (4), 4–7.

Coman, I., Sillitti, A., Succi, G., 2009. A case-study on using an automated in-process software engineering measurement and analysis system in an industrial environment. In: Proceedings of the 31st International Conference on Software Engineering.

Czerwonka, J., Nagappan, N., Schulte, W., Murphy, B., 2013. CODEMINE: building a software development data analytics platform at Microsoft. IEEE Softw. 30 (4), 64–71.

Hammouda, I., Mikkonen, T., Oksanen, V., Jaaksi, A., 2010. Open source legality patterns: architectural design decisions motivated by legal concerns. In: Proceedings of MindTrek'10.

Johnson, P., 2007. Requirement and design trade-offs in Hackystat: an in-process software engineering measurement and analysis system. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement.

Jones, C., Bonsignour, O., 2011. The Economics of Software Quality. Addison-Wesley.

Jones, C., 2008. Applied Software Measurement: Global Analysis of Productivity and Quality, 3rd ed. McGraw-Hill Osborne.

Kanarakus, C., 2013. The worst IT project disasters of 2013. InfoWorld http://www.infoworld.com/d/applications/the-worst-it-project-disasters-of-2013-232402.

Kaur, R., Sengupta, J., 2011. Software process models and analysis on failure of software development projects. Int. J. Sci. Eng. Res. 2 (2), 1–4.

Kersten, M., Murphy, M.G.C., 2006. Using task context to improve programmer productivity. In: Proceedings of SIGSOFT '06/FSE-14.

MacCormack, A., Rusnak, J., Baldwin, C., 2006. Exploring the structure of complex software designs: an empirical study of open source and proprietary code. Manage. Sci. 52 (7), 1015–1030.

Maletic, J., Mosora, D., Newman, C., Collard, M.L., Sutton, A., Robinson, B., 2011. MosaiCode: visualizing large scale software. In: 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis.

Mathews, S., Datar, V., 2007. A practical method for valuing real options: the Boeing approach. J. Appl. Corp. Finance 19 (2), 95–104.

MESA, 1997. White Paper 6, MES Explained: A High Level Vision. MESA Organization.

Mockus, A., Fielding, R., Herbsleb, J., 2000. A case study of open source software development: the Apache server. In: Proceedings of the 22nd International Conference on Software Engineering.

Montaser, A., 2013. Automated Site Data Acquisition for Effective Project Control. Concordia University.

Naedele, M., Kazman, R., Cai, Y., 2014. Making the case for a manufacturing execution system for software development. Commun. ACM 57 (12), 33–36.

Nord, R., Barbacci, M., Clements, P., Kazman, R., Klein, M., 2003. Integrating the Architecture Tradeoff Analysis Method (ATAM) with the Cost Benefit Analysis Method (CBAM). Carneige Mellon University Software Engineering InsituteInstitute Technical Report CMU/SEI-2003-TN-038.

Scacchi, W., Alspaugh, T., 2012. Understanding the role of licenses and evolution in open architecture software ecosys-tems. J. Syst. Softw. 85 (7), 1479–1494.

Schwanke, R., Xiao, L., Cai, Y., 2013. Measuring architecture quality by structure plus history analysis. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 891–900.

Shalal-Esa, A., 2014. Pentagon report faults F-35 on software, reliability. Reuters http://www.reuters.com/article/2014/01/23/us-usa-lockheed-fighter-idUSBREA0M1L920140123.

Standard ANSI/ISA 95.03-2005, 2007. Enterprise-control system integration – Part 3: Activity models of manufacturing operations management, also available as ISO/IEC 62264-3, IEC.

Xiao, L., Cai, Y., Kazman, R., 2014. Design rule spaces: a new form of architecture insight. In: Proceedings of the International Conference on Software Engineering (ICSE) 2014, Hyderabad, India.

ZVEI, 2011. Manufacturing execution systems – industry specific requirements and solutions, www.zvei.org/automation/mes.

Dr. Yuanfang Cai is an Associate Professor at Drexel University. Her research areas include software maintenance, software modularity, software economics, and sociotechnical congruence. Her recent work focuses on assessing software architecture quality based on the synergy of software artifacts, evolution history and the associated team structure.

Dr. Hong-Mei Chen is a Professor of Information Technology Management at the University of Hawaii. Her current research interests include service science/engineering, social (or electronic) customer relationship management, big data, ultra large scale information systems, green information systems and crowdsourced systems. She has directed several large-scale multi-million dollar, multi-institution research projects in high performance telemedicine, multimedia distributed databases, data mining and data warehousing.

Dr. Rick Kazman is a Professor at the University of Hawaii and a Principal Researcher at the Software Engineering Institute. His primary research interests are software architecture, design and analysis tools, software visualization, and software engineering economics. He is the author of over 100 papers, and co-author of several books, including "Software Architecture in Practice", and "Evaluating Software Architectures: Methods and Case Studies".

Dr. Martin Naedele is the Global Head of Architecture for the Business Unit Network Management at ABB. He received a MSEE from Ruhr-University Bochum and a PhD in computer engineering from ETH Zurich. His research interests include software architecture, software engineering, embedded real-time systems, and IT security.

Mr. Carlos V. A. Silva is a graduate student at the University of Hawaii. His research focus is on establishing quantitative and qualitative patterns between structural complexity code metrics and indirect domain dependent measures of effort.

Ms. Lu Xiao is a PhD candidate studying at Drexel University, advised by Dr. Cai. Her research focuses on new representations of software architecture, and associated evaluation techniques for early defect and architecture degradation detection.