Impact of Classes of Development Coordination Tools on Software Development Performance: A Multinational Empirical Study

AMRIT TIWANA
Iowa State University

Although a diverse variety of software development coordination tools are widely used in practice, considerable debate surrounds their impact on software development performance. No large-scale field research has systematically examined their impact on software development performance. This paper reports the results of a multinational field study of software projects in 209 software development organizations to empirically examine the influence of six key classes of development coordination tools on the efficiency (reduction of development rework, budget compliance) and effectiveness (defect reduction) of software development performance.

Based on an in-depth field study, the article conceptualizes six holistic classes of development coordination tools. The results provide nuanced insights—some counter to prevailing beliefs—into the relationships between the use of various classes of development coordination tools and software development performance. The overarching finding is that the performance benefits of development coordination tools are contingent on the salient types of novelty in a project. The dimension of development performance—efficiency or effectiveness—that each class of tools is associated with varies systematically with whether a project involves conceptual novelty, process novelty, multidimensional novelty (both process and conceptual novelty), or neither. Another noteworthy insight is that the use of some classes of tools introduces an efficiency-effectiveness tradeoff. Collectively, the findings are among the first to offer empirical support for the varied performance impacts of various classes of development coordination tools and have important implications for software development practice. The paper also identifies several promising areas for future research.

Categories and Subject Descriptors: D.2.9 [**Software Engineering**]: Management—*Productivity*

General Terms: Design

Additional Key Words and Phrases: Software development, outsourcing, field study, empirical study, regression analysis, software process improvement, knowledge management, development tools, software process improvement, project management, efficiency effectiveness tradeoff, knowledge integration, coordination, collaborative software engineering. Software outsourcing, development coordination tools.

The support of the Russian Development Alliance, the Embassy of Ireland in Washington DC, the Ireland Development Agency, India's National Association of Software and Service Companies, and Emory University is gratefully acknowledged.
Author's address: Iowa State University, 2340 Gerdin, Ames, IA 50011-1350.

ACM Reference Format:
Tiwana, A. 2008. Impact of classes of development coordination tools on software development performance: A multinational empirical study. ACM Trans. Softw. Engin. Method. 17, 2, Article 11 (April 2008) 47 pages. DOI = 10.1145/1348250.1348257 http://doi.acm.org/10.1145/1348250.1348257

1. INTRODUCTION

As the scope, complexity, and geographical dispersion of software development increases, various development coordination tools are being widely adopted in the industry. Development coordination tools are defined as software process support tools that help maintain accuracy, consistency, and integrity of the various mutually interdependent artifacts that are created during the software development process. These artifacts include requirements, architectural views, source code files, test plans, documentation, and reports.

The implicit assumption underlying the adoption of such tools is that they improve software development performance by helping manage the growing complexity of software projects [Messerschmitt and Szyperski 2003]. There is, however, an ongoing debate about whether such tools actually improve software development performance [Brown 2003; Spinellis 2005]. The scant anecdotal evidence that exists suggests mixed results. Others have argued that the usefulness of a particular tool depends on the particular project and that using an inappropriate tool can actually decrease software development performance [Seeley 2003]. Some cautious companies therefore have simply avoided extensively adopting such tools, subscribing to the perspective that it is wiser to not use a tool than to use a "wrong" tool [Brown 2003; Spinellis 2005]. Curiously, the reluctance of software practitioners to extensively adopt such tools has partly been blamed for the declining productivity of the software industry [Groth 2004].

Understanding these issues is important because software developers are often faced with the decision about whether and what combination of development coordination tools to use for any given project [Groth 2004; Seeley 2003]. The ambiguity about the impact of development coordination tools on software development performance can be attributed to the surprising absence of any large-scale industrial studies that have rigorously assessed their individual or simultaneous impact on software development performance in organizations that actually use them. The few studies that have examined the impact of development coordination tools have either focused on specific, proprietary tools within narrowly defined categories such as requirements traceability [Ramesh 1998; Ramesh and Dhar 1992] or have been too broad, focusing on generic conceptualizations of computer aided software engineering (CASE) tools [Guinan et al. 1997; King and Galliers 1994; Orlikowski 1993]. Since the classes of functionality provided by CASE tools varies from one implementation to another, fully understanding their utility also requires delineation of whether they are front-end (focused on the initial design stages of development), back-end (focused on the coding and testing stages of development), or integrated CASE tools (spanning the entire software development life cycle). Prior research on development coordination tools can therefore be characterized as either being too narrowly focused on proprietary tools or too broadly focused on CASE tool collections. Therefore, there exists little generalizable knowledge of how the individual classes of development coordination tools used in software development practice impacts software development performance in actual projects relative to projects that do not use them.

In this article, we attempt to address this gap through a multinational field study, focusing on two research questions.

(1) How does the use of various classes of development coordination tools impact software development performance (efficiency and effectiveness)?
(2) How does their influence vary across projects with different combinations of project novelty (routine, conceptual novelty, process novelty, and multidimensional novelty)?

To address these questions, we conducted a multiphase international field study involving 209 *outsourced* application development projects. We chose to focus on outsourced projects because: (1) they are more likely to require greater coordination across the client and vendor organizations during the development process, (2) specialized, software vendors have historically been early adopters of software development tools, and (3) an increasing proportion of applications development work is outsourced. The software projects in this study were completed for American client companies by software development companies belonging to the three major global software consortia in India, Ireland, and Russia. In the initial phases of the study, we extensively interviewed 26 software practitioners and experts in four countries to identify a field-based classification of six distinct classes of development coordination tools: (1) requirements management tools, (2) architectural modeling tools, (3) test automation tools, (4) test case development tools, (5) configuration management tools, and (6) defect and change request tracking tools. We used these conceptualizations to conduct an extensive industrial field study in which we collected primary data from 209 software development organizations and their clients. We then used this data to examine empirically the relationships between the extent of usage of these classes of development coordination tools and both efficiency (cost and rework) and effectiveness (defects remaining at the time of system delivery) dimensions of software development performance across all projects. We conducted additional, finer-grained analysis to assess how the performance effects of the six classes of development coordination tools vary based on whether a project involves conceptual novelty, process novelty, neither type of novelty (routine), or multidimensional novelty.

Two overarching themes in our empirical findings are noteworthy. First, the usefulness of each class of development coordination tool is contingent on the *combinations* of novelty associated with the project to which it is applied. Second, there is often an efficiency-effectiveness tradeoff in using such tools. Many classes of tools that are associated with improvements in software development efficiency in a given type of project (e.g., reducing rework, and preventing cost overruns) are simultaneously associated with decreases in software development effectiveness. This pattern varies systematically with project novelty *type*. Some of the detailed findings counter popular, taken-for-granted beliefs: Some classes of tools that benefit one type of project can impede another type, challenging the conventional assumption that adopting more development tools can only enhance the software process. Our results show that sometimes a class of tools can be good at what it does but ineffective at solving the knowledge coordination problems specific to a given type of project. Overall, these results have significant implications for choosing an optimal mix of development coordination tools for different types of projects.

The rest of the article is organized as follows. In the next section, we discuss the dominant knowledge integration challenges under each combination of novelty and the role of development coordination tools in overcoming them. We also describe six major classes of tools based on the in-depth qualitative front-end phase of the field study. In Section 3, we formulate a testable formal model. Section 4 describes the field study methodology, including the initial conceptual development interviews, the large-scale, multinational survey, and follow-up interviews. Section 5 presents the statistical analysis and results. Their implications for research and software development practice are discussed in Section 6. The paper concludes with a summary of the key findings.

2. CONCEPTUAL DEVELOPMENT

The central ingredients of software are: (1) technical knowledge through which the software is designed and developed and (2) knowledge of the problem application domain [Adelson and Soloway 1985; Robillard 1999; Rus and Lindvall 2002]. Effective software development requires integration of these two types of knowledge during the development process [Faraj and Sproull 2000; Tiwana and McLean 2003; Walz et al. 1993]. Such knowledge integration is critical for successful software development because it ensures that the design and implementation decisions made on either side of the client-developer organization interface are mutually consistent [Mookerjee and Chiang 2002; Seaman and Basili 1998; Tiwana and Keil 2004]. Effective knowledge integration therefore ensures that the application domain—design dependencies are effectively managed and that what is built closely matches what is needed [Messerschmitt and Szyperski 2003]. The challenge in accomplishing such knowledge integration in outsourced projects arises from the dispersion of these two types of knowledge across the client and software development organizations. The knowledge flow from the client to the vendor and from the vendor to the client that facilitates knowledge integration during the software development process can be viewed as a two-pronged knowledge transfer loop. This is illustrated in Figure 1 and discussed next.

Knowledge about the application domain and project requirements that is needed for design decisions in the software development organization is located in the client organization, but must be accurately transferred to the software development organization [Gottesdiener 2003; Tiwana and McLean 2003; Tiwana and McLean 2005]. Client to vendor knowledge transfer—for example, through requirements and project specifications—represents the *feed-forward*

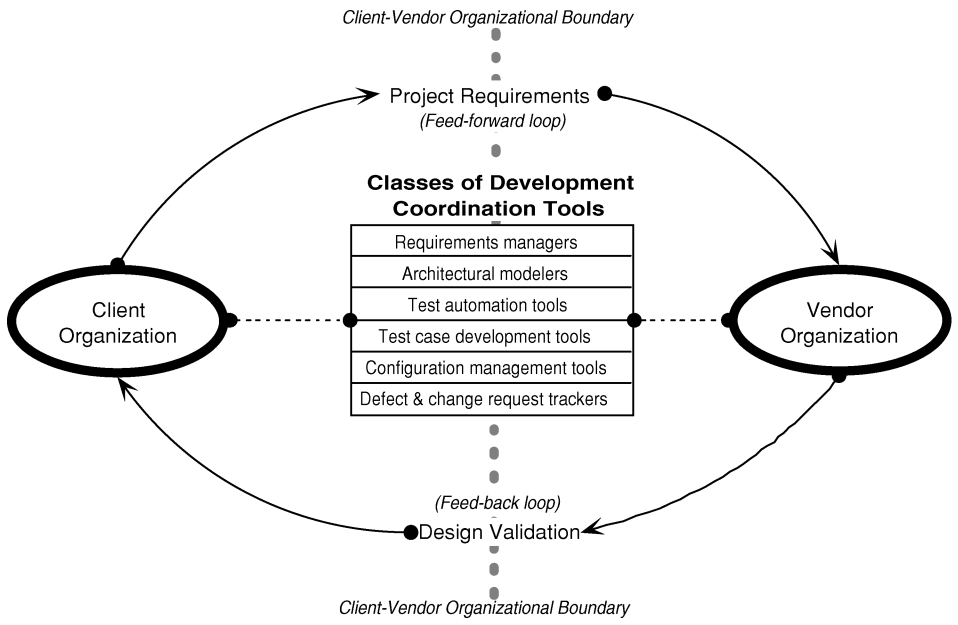
Client-Vendor Organizational Boundary
Project Requirements
(Feed-forward loop)
Classes of Development Coordination Tools
Requirements managers
Architectural modelers
Test automation tools
Test case development tools
Configuration management tools
Defect & change request trackers
Client Organization
Vendor Organization
(Feed-back loop)
Design Validation
Client-Vendor Organizational Boundary

Fig. 1. Various classes of development coordination tools can facilitate coordination of requirements and design validation activities across the client-vendor organizational boundary/ interface.

portion of this loop. Solely for simplicity of discussion, we label this facet of knowledge transfer between the client and vendor as the project *requirements loop*. However, it is well recognized that initial project requirements and specifications are imperfect and insufficient knowledge transfer mechanisms [Messerschmitt and Szyperski 2003]. Therefore fully incorporating the client's intended objectives, constraints, and requirements missed in the preliminary formal requirements requires iterative refinement. Thus information about the in-progress design that is generated in the software development organization must be validated by the client, requiring transferring it from the vendor to the client [Abdel-Hamid et al. 1993; Ramesh and Dhar 1992]. Such iterative refinement and validation of the preliminary project requirements requires the complementary feedback facet of the knowledge transfer loop. We label this feedback loop the project *design validation loop*. The requirements loop and the design validation loop therefore represent the two complementary *feed-forward* and *feed-back* directions of knowledge transfer between the client and vendor organizations during the software development process.

The role of various classes of development coordination tools can be analyzed by considering the loop—requirements or design validation—in which problems are more likely to arise under different combinations of novelty. The classes of development coordination tools that are most likely to improve software development performance are those that facilitate overcoming the knowledge integration problems *specific to that type of project*. The prior literature on development coordination tools is, however, very limited and does not provide theoretical guidance on this issue. As noted earlier, much of the attention in that body of work has been focused on specific tools rather than classes of tools and almost none of the prior work has examined the relationships between tool class usage and software development performance. For example, prior work has developed the general implementation of a requirements traceability tool such as REMAP [Ramesh and Dhar 1992; Ramesh and Jarke 2001]. It however remains unclear how this tool generalizes to other requirements management tools. Such emphasis on specific implementations of tools makes it difficult to generalize to the broader performance implications of development coordination tools unless we move away from the numerous, specific tools and abstract towards the core characteristics and the underlying functionality of generic, holistic classes of tools. On the other extreme, other prior work has examined the adoption (but not the performance impacts) of computer-aided software engineering (CASE) tools [Orlikowski 1989, 1993]. CASE tools provide a plethora of coordination functionality and such functionality often varies from one CASE tool package to another. For example, some CASE tools support early stages of the software design process (front-end CASE tools), others focus primarily on the later development and testing stages (back-end CASE tools), while others support the entire software development lifecycle (integrated CASE tools). This coarse level of granularity makes it impossible to draw generalizable inferences about their influence on software development performance because different CASE tools represent different bundles of development coordination functionality (which makes it difficult to compare across projects or organizations).

To move past this dual problem of high specificity and lack of granularity in the treatment of development coordination tools in prior work, we attempted to abstract the distinct types of functionality that different classes of development coordination tools can provide through an extensive set of in-depth interviews in 19 software development companies. This step was a necessary antecedent to the subsequent empirical field study. Because without a common set of development coordination tool definitions, it would be impossible to draw generalizable insights about development coordination tool functionality and software development performance given the diverse variety of proprietary and commercial tools that software development organizations used.

We developed an abstracted list of development coordination tool classes using a grounded, semi-structured interview methodology (Phase 1 in Figure 2) [Creswell 1994; Morgan and Smircich 1980; Strauss and Corbin 1998]. Grounded field research refers an inductive (observations $\rightarrow$ to $\rightarrow$ theory) approach where observations from the field are used to generate conceptual categories. (An in-depth discussion of the methodology appears in the next section.) This involved three rounds of interviews with software project managers in 19 software development organizations and seven academic software engineering and project management experts. These interviews were used to identify the most commonly used classes of development coordination tools through qualitative pattern mapping [Todd 1979] using a grounded conceptual development approach [Baskerville and Pries-Heje 1999; Glaser and Strauss 1967].

The key classes of development coordination tools commonly used in practice were identified and preliminary descriptions of these tools were created through this process. These descriptions were then refined following extensive feedback from these software practitioners to ensure that the descriptions of the six classes tools provide to the field study participants in the subsequent questionnaire were interpreted consistently and without ambiguity [Mumford et al. 1996; Straub 1989]. The study participants commented on the clarity and meanings of the tool class descriptions, definitions, and conceptualizations. In our interviews, we found that there was variance in the tools that different organizations used to accomplish similar functions. Abstracting generalized classes and descriptions of the key classes of development coordination tools used in practice was the most critical yet challenging aspect of the questionnaire development process. We resolved this challenge by moving away from specific implementations of tools to instead providing descriptions of the functionality provided by each class of development coordination tool. The final version of the descriptions of the functionality of each class of development coordination tools based on these iterative interviews is summarized in Table I. This grounded conceptual development process helped establish the face validity and content validity of the description of various classes development coordination tools that were used in the subsequent questionnaire-based data collection phase of the study. This questionnaire draft was pretested once again to ensure that the respondents correctly and unambiguously interpreted the meaning of each class of tools. This iterative refinement process was repeated until further improvements in the questionnaire appeared to be marginal [Straub 1989]. Six salient classes of development coordination tools emerged from this phase of the study.

We focus our attention on these six classes of development coordination tools: (1) requirements management tools, (2) architectural modeling tools, (3) test automation tools, (4) test case development tools, (5) configuration management tools, and (6) defect & change request tracking tools. To the extent that a class of tools facilitates integration of fragmented technical and application domain knowledge in a given type of project, it is likely to enhance software development performance.

The salient knowledge integration challenges in routine (see Section 2.1), conceptually novel, process novelty, and multidimensionally novel (see Section 2.2.) projects are discussed next (see Table II for a summary). This discussion is then used to theorize how different classes of development coordination tools might help overcome these challenges.

2.1 Routine Projects

Routine projects represent a well-definable problem space for which the initial state, the goal, and a set of possible operations to reach the goal from the initial state are available [Robillard 1999]. Routine projects are therefore low in conceptual and process novelty. Here, formal requirements facilitate conveying client needs to the software development organization. In routine projects, client requirements are more likely to be known with some confidence at the outset of the project and interactions between the system and other subsystems in the client organization can be identified early on. Further, the use of

Table I. The Six Classes of Development Coordination Tools and Their Roles in the Software Process

Class of Development Coordination Tools	Definition and functionality provided	Role in knowledge integration during software development
Requirements managers	Maintaining system and subsystem requirements and their interdependencies.	Maintaining integrity between the systems design and knowledge of evolving client needs.
Architectural modelers	Maintaining various modeling views and high-level design information.	Translating client requirements into a high-level software design that accurately reflects those requirements.
Test automation tools	Executing the test cases and generating defect reports.	Increasing the efficiency with which the mapping between client requirements and the in-progress system artifacts can be frequently assessed; lowering the overhead of testing the integrity of system artifacts.
Test case development tools	Helping create system-level test-case descriptions that confirm whether the known requirements are met.	Ensuring that knowledge of client needs as specified in project requirements is integrated in the system being developed.
Configuration managers	Keeping track of various versions of all of the project's artifacts and providing a mechanism for associating project artifact versions and the delivered version of the system.	Tracing the lifecycle of systems functionality to prior versions of the underlying design artifacts and to the underlying client requirements.
Defect & change request trackers	Allowing any stakeholder to report problems or suggest changes to the system through a central repository. Defining and enforcing business rules and decision authority allocation for incorporating various types of changes.	Providing a centralized mechanism for keeping track of changes requested by project stakeholders and enforcing authority for making different types of changes across the development organization and client organization.

established development processes facilitates validation of the design by the client organization while development is in progress [Kokol 1989].

Many of the classic project management concepts for software development embody models that portray development as a sequential process of design followed by development and testing [MacCormack et al. 2001]. Effective software

Table II. The Dominant Knowledge Integration Challenges in Different Types of Projects

Project Type	Loop in which dominant knowledge integration challenge exists (see Figure 1)
Routine	No dominant knowledge integration challenge exists and the utility of development coordination tools is primarily improvement-oriented.
Conceptual novelty	REQUIREMENTS LOOP i.e., accurately conveying client organization needs to the software development organization.
Process novelty	DESIGN VALIDATION LOOP i.e., ensuring that the delivered system matches the client's needs.
Multidimensional novelty	Challenges in knowledge integration across REQUIREMENTS LOOP and DESIGN VALIDATION LOOPS simultaneously exist.

development in these models is characterized by an approach that minimizes changes to the design once coding has begun. A strong assumption in such models is that all information about potential design choices is known or can be readily discovered during concept development. Projects that either attempt to develop a solution that is novel in concept or attempt to follow a new development process present a fundamental challenge to such asserted models of software development. A second challenge that this oversimplified view faces is that feedback on the performance and functionality of the system is not obtained until much later in the development process, when a large proportion of the functionality has already been implemented. These are relatively less challenging assumptions in routine projects.

Even though no *dominant* knowledge integration challenge exists in routine projects, development coordination tools can enhance knowledge integration across the feed-forward knowledge transfer loop by helping better translate project requirements into a high level software design (e.g., using architectural modeling and requirements modeling tools) and the feed-back knowledge transfer loop by helping verify through appropriate test cases that this has been acceptably accomplished (e.g., using defect/change request tracking, test case development, and test automation tools). We therefore expect these all six classes of development coordination tools will be associated with enhanced development performance in routine projects.

2.2 Novel Projects

Knowledge Integration Challenges Common to All Novel Projects. In the presence of any type of novelty, the most challenging aspect of knowledge integration is the client organization's inherent difficulty in precisely spelling out their novel project concepts through formal requirements at the outset of the project (i.e., what we labeled the *requirements loop*) [Rowen 1990]. The most likely error in such projects is that initial, formal project requirements do not fully capture the client's needs and constraints [McAfee 2003]. This can happen for two possible reasons. First, the initial requirements might have been incomplete because it is much more difficult to comprehensively articulate and formally transfer such knowledge (e.g., in the form of project requirements) to the software development organization [Tiwana and McLean 2005]. Second, as the client organization begins to see preliminary development artifacts such as architecture representations, modeling diagrams, prototypes, and mock-ups, it may recognize that the software development organization's understanding of the client problem domain is incomplete or imprecise. This can introduce new requirements during development. The software development organization must integrate such emerging information about client needs *during the development process* to avoid the risk of proceeding to develop a system that fails to meet the client's needs.

We therefore expect that requirements management tools, which facilitate managing requirements and interdependencies among requirements as they evolve (or become clearer) over the project's lifecycle will be associated with higher software development performance in all types of novel projects. In addition, once test cases are developed, automated testing of the code throughout the development lifecycle lowers coding errors, thus can be expected to be associated with higher software development performance across all types of novel projects.

Knowledge Integration Challenges Unique to Conceptually Novel Projects. The dominant challenge in conceptually novel projects lies in accurately conveying the novel client requirements from the client organization to the software development organization (i.e., across the REQUIREMENTS LOOP). Therefore, coordination tools that facilitate associating evolving client requirements with project artifacts (requirements management and defect/change request tracking tools) and maintain traceability across artifact versions and system functionality (e.g., configuration management tools) can enhance software development performance. Because requirements are more likely to evolve over the course of conceptually novel projects, there is inherent value in being able to forward-trace and backward-trace the evolution of different versions of various project artifacts such as requirements, specifications, and the software modules associated with them (configuration management tools). Furthermore, classes of tools that facilitate testing (test case development and automation tools) can also enhance software development performance by freeing up development resources ordinarily dedicated to rote testing activities to instead focus on the conceptually new problem space. Automating the execution of test cases and frequent generation of test reports throughout the development process can therefore lower the likelihood of cost overruns. Paradoxically, test automation development coordination tools can increase the efficiency of intensive and frequent testing but might also lower the likelihood of *finding* and subsequently removing more defects in conceptually novel projects (especially if the test cases are inadequate), which might be reflected in lower development rework and costs. Architectural modeling tools however can only limitedly enhance development performance because the preliminary project requirements that serve as inputs for constructing architectural models might themselves be incomplete at the outset. Except for these, we therefore expect all other classes of development coordination tools to be associated with improvements in at least one dimension of performance in conceptually novel projects.

Knowledge Integration Challenges Unique to Projects Involving Process Novelty. Use of novel development processes (such as a proprietary methodology mandated by the client organization or a development process with which the software development organization is not familiar) means that there is no longer an established, mutually understood "syntax" through which the client can monitor and provide feedback to the software development organization during development. Although the uncertainties might be manageable from a technical standpoint, the use of novel development processes makes it more difficult to verify that the in-progress design is compatible with the needs of the client organization and reflects an architecture that best captures the desired functionality and feature tradeoffs. It is useful to think in terms of degree of process novelty instead of thinking of it in dichotomous terms. Therefore, projects that require the use of many new processes at once or processes that span several different parts of an organization have higher process novelty relative to one that uses a minor modification of an existing development process. Lacking a preestablished process for validation, the dominant challenge in projects involving process novelty lies in validating that the artifacts generated by the vendor correspond to the client organization's actual needs (i.e., the DESIGN VALIDATION LOOP). The most effective classes of development coordination tools in such projects are the ones that help: (a) integrate information about the in-progress project artifacts generated in the software development organization for client design validation (requirements management tools), and (b) manage interdependencies among client requirements, versions of project artifacts, and system modules, which might be more difficult to establish without the use of preexisting development processes (e.g., configuration management tools). This allows the development team to trace the evolution of, say a requirement or a user interface feature, without being overly dependent on the procedural routines of the novel process. However, since a well-established, familiar software process that both the client and vendor are familiar with does not exist in projects with process novelty, successfully accomplishing the automation of specialized development activities using such test case development tools and change request trackers is likely more difficult. Therefore, we expect limited benefits to be realized from the use of such classes of tools.

Knowledge Integration Challenges in Projects Simultaneously Involving Conceptual and Process Novelty (Multidimensional Novelty). Projects with multidimensional novelty face challenges common to both conceptually and procedurally novel projects because both types of novelties simultaneously exist in the project. Therefore, the classes of development coordination tools that facilitate knowledge integration across both the requirements loop and design validation loop are likely to be the most beneficial. Classes of tools that facilitate knowledge integration in the REQUIREMENTS LOOP (e.g., requirements and defect/change request tracking tools) as well as the DESIGN VALIDATION LOOP (test case development and automation tools that facilitate efficient and frequent execution of test cases and defect report generation) can thus be expected to enhance software development performance. Architectural modeling tools however, encounter the same bottlenecks as in conceptually novel projects, that is, lack of a stable and

Table III. Summary of Development Coordination Tools Classes that Were Expected to Influence Software Development Performance

	Project Novelty Type			
Class of Tools	Routine	Conceptual	Process	Multidimensional
Requirements managers	•	•	•	•
Architectural modelers	•			
Test automation tools	•	•	•	•
Test case development tools	•	•		
Configuration management tools	•	•	•	•
Defect & change request trackers	•	•		•

complete set of preliminary requirements on which to base the high-level design. This coupled with the lack of an established process makes it more difficult to manage interdependencies among client requirements and versions of project artifacts, limiting the benefits realized from using configuration management tools.

In summary, although using various classes of development coordination tools can enhance software development performance in routine projects, their value in other types of projects is contingent on the type(s) of novelty present in the project. The influence of a specific class of development coordination tools will be more pronounced when its use facilitates integrating the type of distributed knowledge that is problematic in that *type* of project. A class of tools that might enhance the development process for one type of project might yield no observable benefits for other types. Clearly, the dimension of development performance that each class of development coordination tool enhances in each type of project is an empirical question. The foregoing discussion is summarized in Table III. The bullets indicate whether the each class of tool is expected to be associated with an improvement in software development performance for each type of project.

In the following section, we use the foregoing discussion to develop an empirically testable model for assessing the impact of each class of development coordination tools on software development performance.

3. MODEL SPECIFICATION

A system of regression equations in which software development performance is modeled as a function of the level of use of each of the six classes of tools (defined later in this section) was first developed. These were assessed empirically using data from the projects in the study to test the relationships between tool class usage and software development performance. The objective of the formal models is twofold: (1) to assess whether each of the six classes of development coordination tools significantly influences rework, cost overrun, and residual defects and (2) to assess the direction and *relative* magnitude of this relationship in routine, conceptually novel, procedurally novel, and multidimensionally novel projects.

The model is specified as a four-step hierarchical regression model where performance is a function of the usage of the six classes of tools. Three such models are estimated, one each for the three dependent variables—development rework, cost overrun, and residual defects. The first two represent proxies for the efficiency dimension and the third the effectiveness dimension of software development performance. *Development rework* refers to the percentage of project function points that were changed during the testing stages of a project. This represents development effort that was expended after the main development stages of the project that ideally should not have been expended at all if the coded system were perfect at the time of its delivery to the client organization. *Cost overrun* refers to the percentage by which a project exceeded its planned budget. *Residual defects* refer to the defects recorded in the post-delivery installation and warranty stages of a project. We chose to focus on these three dependent variables because budget compliance, development effort and rework, and quality (measured as the absence of defects at the time of delivery) are widely used metrics for software development performance.

Each equation specifies the dependent variable as a function of the extent of use of the six classes of development coordination tools in a project. Each equation represents a four-stage, incremental stepwise regression model. Each of the four steps tests the effect of the six classes of development coordination tools incrementally in routine projects followed by conceptual, process, and multidimensional novelty. The corresponding equation for the first dependent variable *rework* is summarized in Equations (1) through (4), as follows. In the first regression-modeling step, the usage data on the six coordination tool classes were entered into the model. (ε *r*epresents the error term in the regression equation.)

In the equations that follow, the variables represent the following:

$RMgr$ = *requirements management tool usage level*

$Modeling$ = *architectural modeling tool usage level*

$Test\_auto$ = *test automation tool usage level*

$Test\_case$ = *test case development tool usage level*

$Config$ = *configuration management tool usage level*

$Tracking$ = *defect and change request tracking tool usage level*

$Cnew$ = *conceptual novelty*

$SPNew$ = *software process novelty*

$MDNew$ = *multidimensional novelty*

$$\begin{aligned} Rework \;=\; & \beta_0 + \beta_{rmgr}\mathrm{RMgr} + \beta_{modeling} Modeling \\ & + \beta_{test\_auto} test\_auto + \beta_{test\_case} Test\_case \\ & + \beta_{config} Config + \beta_{tracking} Tracking + \varepsilon \end{aligned} \tag{1}$$

Next, the effects of each class of development coordination tools in the presence of conceptual novelty were assessed by adding six interaction terms between conceptual novelty and tool class usage.

$$\begin{aligned} & + \beta_{rmgr\text{-}c}\mathrm{RMgr}^{*} Cnew + \beta_{modeling\text{-}c} Modeling^{*} Cnew \\ & + \beta_{test\_auto\text{-}c} test\_auto^{*} Cnew + \beta_{test\_case\text{-}c} Test\_case^{*} Cnew \end{aligned}$$

$$+ \beta_{config\text{-}c} Config^{*}Cnew + \beta_{tracking\text{-}c} Tracking^{*}Cnew. \qquad (2)$$

Next, the effects of each class of development coordination tools in the presence of software process novelty were assessed by adding six interaction terms between process novelty and tool class usage. This interaction terms approach allowed a finer grained assessment using *levels* of each type of novelty as opposed to a simple binary classification approach that would inherently be less granular.

$$\begin{aligned}
&+ \beta_{rmgr\text{-}sp} RMgr^{*}SPNew + \beta_{modeling\text{-}spg} Modeling^{*}SPNew\\
&+ \beta_{test\_auto\text{-}sp} test\_auto^{*}SPNew + \beta_{test\_case\text{-}sp} Test\_case^{*}SPNew\\
&+ \beta_{config\text{-}sp} Config^{*}SPNew + \beta_{tracking\text{-}sp} Tracking^{*}SPNew \qquad (3)
\end{aligned}$$

Finally, the effects of each development coordination tool class in the presence of multidimensional novelty were assessed by adding six interaction terms between tool class usage and multidimensional novelty (a dummy-coded variable) involved in the projects.

$$\begin{aligned}
&+ \beta_{rmgr\text{-}md} \mathrm{RMgr}^{*}MDNew + \beta_{modeling\text{-}mdg} Modeling^{*}MDNew\\
&+ \beta_{test\_auto\text{-}md} test\_auto^{*}MDNew + \beta_{test\_case\text{-}md} Test\_case^{*}MDNew\\
&+ \beta_{config\text{-}md} Config^{*}MDNew + \beta_{tracking\text{-}md} Tracking^{*}MDNew \qquad (4)
\end{aligned}$$

The final model for rework then becomes the following.

$$\begin{aligned}
&Rework = \beta_0 + \beta_{rmgr}\mathrm{RMgr} + \beta_{modeling} Modeling\\
&+ \beta_{test\_auto} test\_auto + \beta_{test\_case} Test\_case + \beta_{config} Config\\
&+ \beta_{tracking} Tracking + \beta_{rmgr\text{-}c}\mathrm{RMgr}^{*}Cnew\\
&+ \beta_{modeling\text{-}c} Modeling^{*}Cnew\\
&+ \beta_{test\_auto\text{-}c} test\_auto^{*}Cnew + \beta_{test\_case\text{-}c} Test\_case^{*}Cnew\\
&+ \beta_{config\text{-}c} Config^{*}Cnew + \beta_{tracking\text{-}c} Tracking^{*}Cnew\\
&+ \beta_{rmgr\text{-}sp}\mathrm{RMgr}^{*}SPNew + \beta_{modeling\text{-}spg} Modeling^{*}SPNew\\
&+ \beta_{test\_auto\text{-}sp} test\_auto^{*}SPNew + \beta_{test\_case\text{-}sp} Test\_case^{*}SPNew\\
&+ \beta_{config\text{-}sp} Config^{*}SPNew + \beta_{tracking\text{-}sp} Tracking^{*}SPNew\\
&+ \beta_{rmgr\text{-}md}\mathrm{RMgr}^{*}MDNew + \beta_{modeling\text{-}mdg} Modeling^{*}MDNew\\
&+ \beta_{test\_auto\text{-}md} test\_auto^{*}MDNew + \beta_{test\_case\text{-}md} Test\_case^{*}MDNew\\
&+ \beta_{config\text{-}md} Config^{*}MDNew + \beta_{tracking\text{-}md} Tracking^{*}MDNew + \varepsilon \qquad (5)
\end{aligned}$$

Cost overruns and residual defects were predicted by similar regression equations (6), (7) in which the dependent variable was respectively development rework and cost overrun. The underlying logic for these equations mirrors that of the equations for rework.

$$\begin{aligned}
&Costoverrun \parallel Residualdefects = \beta_0 + \beta_{rmgr}\mathrm{RMgr} + \beta_{modeling} Modeling\\
&+ \beta_{test\_auto} test\_auto + \beta_{test\_case} Test\_case\\
&+ \beta_{config} Config + \beta_{tracking} Tracking\\
&+ \beta_{rmgr\text{-}c}\mathrm{RMgr}^{*}Cnew + \beta_{modeling-c} Modeling^{*}Cnew
\end{aligned}$$

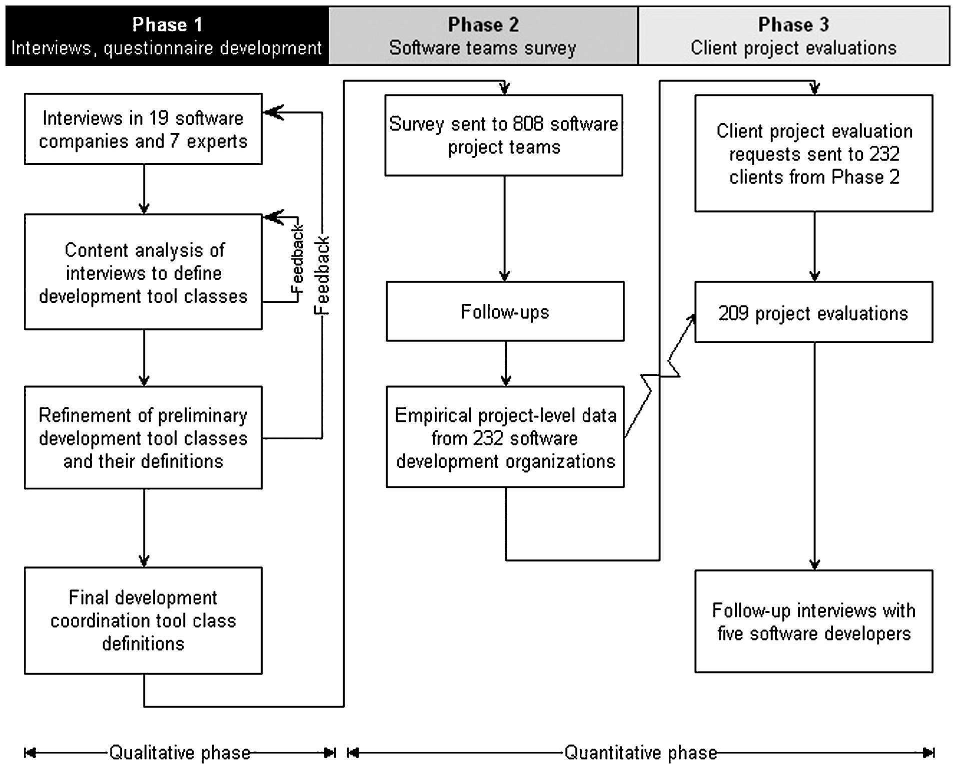
Phase 1
Interviews, questionnaire development
Phase 2
Software teams survey
Phase 3
Client project evaluations
Interviews in 19 software companies and 7 experts
Content analysis of interviews to define development tool classes
Refinement of preliminary development tool classes and their definitions
Final development coordination tool class definitions
Feedback
Feedback
Survey sent to 808 software project teams
Follow-ups
Empirical project-level data from 232 software development organizations
Client project evaluation requests sent to 232 clients from Phase 2
209 project evaluations
Follow-up interviews with five software developers
Qualitative phase
Quantitative phase

Fig. 2. Overview of study methodology.

$$
\begin{aligned}
&+ \beta_{test\_auto-c} test\_auto^{*}Cnew + \beta_{test\_case-c} Test\_case^{*}Cnew\\
&+ \beta_{config-c} Config^{*}Cnew + \beta_{tracking-c} Tracking^{*}Cnew\\
&+ \beta_{rmgr\text{-}sp} \mathrm{RMgr}^{*}SPNew + \beta_{modeling-spg} Modeling^{*}SPNew\\
&+ \beta_{test\_auto-sp} test\_auto^{*}SPNew + \beta_{test\_case-sp} Test\_case^{*}SPNew\\
&+ \beta_{config-sp} Config^{*}SPNew + \beta_{tracking-sp} Tracking^{*}SPNew\\
&+ \beta_{rmgr\text{-}md} \mathrm{RMgr}^{*}MDNew + \beta_{modeling-mdg} Modeling^{*}MDNew\\
&+ \beta_{test\_auto\text{-}md} test\_auto^{*}MDNew + \beta_{test\_case-md} Test\_case^{*}MDNew\\
&+ \beta_{config-md} Config^{*}MDNew + \beta_{tracking-md} Tracking^{*}MDNew + \varepsilon \qquad (6)
\end{aligned}
$$

The regression models were estimated using empirical data collected in the field study and the coefficient estimates and tests for their statistical significance provide insights into which classes of tools provide what performance benefits under each of the four possible types of project novelty.

The regression estimation approach facilitates statistically testing the relationships between tool class usage and development performance in the presence of various combinations of novelty in the following manner. The first modeling step was to assess the impact of development coordination tool class usage in routine projects, that is, without conceptual or process novelty. In the first step, the "main effects" terms for the extent of use of the six classes of development coordination tools are entered in the regression equation. This corresponds to routine projects. Next, six interaction terms between each development coordination tool class and a dummy variable for conceptual novelty is entered in the model. This corresponds to conceptually novel projects. In the third step, six interaction terms between each of the six development coordination tool classes and a dummy variable for process novelty is entered in the model to assess how they relate to development performance in projects with process novelty. In the fourth step, six interaction terms between each development coordination tool class and a dummy variable for multidimensional novelty (simultaneous conceptual and process novelty in the project) is entered in the regression model to assess how they relate to development performance in projects with multidimensional novelty. (To create this dummy variable, we tagged projects that had both conceptual novelty and process novelty scores simultaneously rated at the highest level of the scales as 1 and others as 0.)

4. RESEARCH METHODOLOGY AND DATA COLLECTION

Figure 2 provides an overview of the three phases of the field study methodology used in the study. The field study used a sequential multimethod research design [Mingers 2001]. The "field study" label refers to the context in which the study was conducted, that is, real-world software development organizations. The multimethod approach refers to the use of two methods in a sequential manner. In this study, the upfront phase of the study used a semi-structured, interview-based qualitative methodology (phase 1). The results of the qualitative phases were used to inductively derive "grounded" descriptions of various classes of development coordination tools that were then used to conduct the large-scale industrial survey using quantitative empirical methods [Baskerville and Pries-Heje 1999; Strauss and Corbin 1998].

In phase 1, multiple rounds of in-depth interviews with 26 domain experts (software project managers in 19 software organizations and 7 academic software development experts) were conducted to identify and define the development coordination tools used in practice. Having been drawn from four different countries, the interviewees were representative of the diversity of the eventual sampling frame of companies that were targeted in Phases 2 and 3. This was particularly important since similar concepts might be described differently across software development companies in the four nations that the study spanned. We interviewed software project managers in six U.S. companies, four Indian, two Irish, and seven Russian software development companies in this phase. All interviews were conducted in English. The interviews were conducted in person and over the phone. They followed a semi-structured interviewing protocol outlined by Yin [1994] to elicit identification and description of various classes of development coordination tools used in the interviewees' companies. Content analysis procedures were used to identify questionnaire items to measure development coordination tool usage [Lee 1989], which were then further refined through two rounds of additional interviews [Mingers 2001]. (The details of this procedure were described in detail in the conceptual development section of the paper.) The use of a qualitative front-end to develop the content for field survey is the convention for field-based surveys in organizations in the social sciences including information systems [Mumford et al. 1996; Schwab 1980]. The outcome of the first phase of the study was therefore a set of grounded descriptions of six classes of development coordination tools that could be used unambiguously and reliably to describe each class of tools to a variety of software development organizations. This formed the input for Phase 2 of the study.

In phase 2, project-level empirical data on tool class usage and archival defect data for each project stage were collected from the lead project managers in 232 software development companies in India, Ireland, and Russia. The top executives of 818 software development organizations in the three largest global software consortia in Russia (Russian National Software Development Alliance), Ireland (via Irish Investment and Development Agency), and India (National Association of Software and Service Companies) were requested to identify a major project that their company had completed within the last two years for a U.S. organization. The lead project managers of each of these projects completed the software development organization-side questionnaire and provided archival defect data for each stage including unit-testing, system testing, acceptance testing, installation, and warranty stages of individual projects. In this phase of the study, objective archived data were used where possible and all other constructs in the study were measured on a 7-point Guttmann scales using four to six questionnaire items. A Guttman scale (also known as scalogram analysis) is a perceptual measurement instrument developed using the Guttman scaling technique. A Guttman scale is a cumulative scaling approach for measuring concepts that have varying levels. The primary objective of the Guttman scaling is to ensure that the instrument measures only a single trait (a property described as unidimensionality wherein a single dimension underlies the responses to the scale). The purpose of such a scale is to establish a one-dimensional continuum for a concept being measured. For example, the scale for conceptual novelty has four measurement items. As a respondent moves from the first to the last item of the scale, it indicates that the level of conceptual novelty has cumulatively increased. In a Guttman scale, respondents who agree with a more extreme scale item also agree with all less extreme scale items that precede it.

Data on the extent of use of each of the six classes of development coordination tools were obtained from the lead project manager in each software development organization. Data on the outcomes of each project were obtained from project-manager-reported and client-reported archival development records for each project. This multi-source approach was used to mitigate the threat of common methods bias that arises from obtaining perceptual data for both the dependent and predictor variables from the same respondent in a single survey. Appropriate steps were taken to ensure that the measurement scales measured distinct variables and tapped into the conceptual domain of the variable (i.e., exhibited unidimensionality, face validity, and content validity) [Bagozzi et al. 1991; Mumford et al. 1996; Straub 1989]. Since all multiitem scales (tool class usage) were cumulative Guttman scales and performance variables were each measured as a single item, no alphas [Cronbach and Meehl 1955] are associated with them [Schmidt and Hunter 1989]. The survey questionnaire items for each construct are described in the next section. The data from Phase 2 of the study were used as the basis for collecting additional software development performance data from the client organization in Phase 3.

In phase 3, software development performance assessments of cost overruns were obtained from client liaisons for each of these projects. Finally, after the statistical analysis, we conducted additional interviews with five software developers to validate and interpret some of the empirical findings. This *key informant technique* of collecting data from a highly knowledgeable individual in each organization is a commonly used approach in field studies [Kumar et al. 1993].

The choice of the key informants for the survey was an important consideration. The objective was to collect project-level data from someone who was knowledgeable about all the development coordination tools that were used in a particular project and less likely to be biased towards providing responses that might appear to be socially desirable. Meeting these criteria required collecting data from two different individuals from the client and vendor organizations, further complicating the challenges of collecting field-based data. The lead project manager for each project was an appropriate respondent for each project because he/she is most likely to have good *overall knowledge* of the extent to which different development coordination tools were used in the project. In contrast, project team members might have more knowledge of the tools that they *personally used* in the project, but might not have accurate knowledge of the extent to which ones that they themselves did not use but other project team members did. The client-side liaison managers from the client organization were the most appropriate source of information about deviation from budget because they were most likely to possess budget information as well as less likely to underreport it. In contrast, collecting this information directly from the vendor could have introduced reporting bias. Our preliminary interviews in Phase 1 of the study also confirmed this choice of informants.

4.1 Measures

The following section describes how the key constructs in the study were measured in the survey questionnaire. Development coordination tool usage was measured for each of the six classes of tools using a six-item, seven-point Guttmann scale. The descriptions shown in Table I were provided to the respondents at the beginning of this questionnaire section to ensure that all of them interpreted these tools in a consistent and unambiguous manner. The scale of 1 to 7 measured the extent to which the following classes of tools were used in the development process for each project, based on the lead project manager's assessment in each software development organization. The end points of the scale were "not at all" to "to a great extent" with "somewhat" as an anchor for the midpoint of the scale.

(a) Requirements managers
(b) Architectural modelers

(c) Test automation tools
(d) Test case development tools
(e) Configuration managers
(f) Defect & change request tracking tools

Conceptual novelty (CNew) was measured using a four-item Guttmann scale adapted from [Takeishi 2002]. This scaled measurement approach provides a granular measure for different the levels of novelty, where (1) below is the least conceptually novel and (4) the most conceptually novel. Each level is coded as one through four for statistical analyses. The advantage of this scaled measurement approach over a discrete, binary (novel/not novel) measure is that finer grained analysis is possible through multiple regression modeling and the ability to construct discrete classifications is retained. Both continuum-based values and discrete classification are later used for different facets of the statistical analyses. Information on this variable was collected from the software development organization. The four questionnaire items for this scale asked the software development organization to assess the novelty in the project according to the description that best characterized the project from the following four questionnaire items.

(1) Minor modification of a system design already developed by your company
(2) Major modification of a system design already developed by your company
(3) Completely new design, but based on a concept already demonstrated in another project
(4) Technically new to your company and a completely new design

Software process novelty (SPNew) was measured using a four item Guttmann scale based on Takeishi [2002] and Adler [1995]. Information on this was collected from the software development organization. The four questionnaire items for this scale asked the software development organization to assess the novelty in the project according to the description that best characterized the project from the following four questionnaire items.

(1) Existing methodology and development tools used with MINOR modifications
(2) Existing methodology and development tools used with MAJOR modifications
(3) Either new methodology OR development tools, but based on existing ones
(4) Entirely new methodology AND new development tools

The efficiency dimension of software development performance was measured using rework and cost over run as proxies and the effectiveness dimension of performance was measured using the residual, post-delivery defects that were recorded in the installation and warranty stages of a project as a proxy. Rework was measured as the percentage of the total project function points that were changed during the testing stages of each project. Information on this was collected from the software development organization. Cost overrun was measured as the percentage by which the project exceeded its planned budget allocated by the client organization. Information on this was collected from the primary project-liaison in the client organization. This approach mitigates the methodological threat usually referred to a "common-methods bias" in survey research [Babbie 1973]. This bias arises when the dependent variable data (cost overrun) are collected from the same organization that might be biased to underreport cost overrun, that is, the vendor. This problem does not exist in objective (nonperceptual) variables such as recorded defects counts, total team size counts, and allocated project schedule (in months). Residual defects represent the number of defects recorded in the installation and warranty stages that followed unit-, system-, and acceptance-testing, that is, those that existed in the system at the time of project delivery to the client organization. However, since the number of defects can vary with project size, we accounted for project person-months, which is a reasonable proxy for project size. Although project size could have been measured in thousand line of code (KLOCs), our initial interviews revealed that different companies differed on which of these metrics they internally used. Project person months was therefore a common metric applicable to all of the companies in the sample.

4.2 Response Rates

We received an overall response rate of about 28% and only the 209 projects with matched pair data were used in the statistical analyses. We received completed sets of questionnaires back from 33.5% (59 received/176 sent) of the Russian, 29.5% of the Irish (54 received/183 sent), and 25.9% of the Indian vendors (119 received/459 sent) respectively. 23 of these 232 projects had to be dropped from the sample because of missing matched-pair data, leaving 209 projects for the purpose of our analyses. We attribute the high response rate to targeting software development organizations belonging to three competing international consortia, a comparison report custom-developed for each respondent's company in return for participation, multiple follow-ups, as well as extensive confidentiality agreements. Additional checks suggest no persuasive threat of nonresponse bias (this is discussed in detail later under Threats to Validity). At the end of this phase, we interviewed five software developers to gain additional insights into the results from our preliminary statistical analyses.

5. ANALYSIS AND RESULTS

5.1 Project Characteristics and Descriptive Statistics

The projects in the study were custom application development projects. The key characteristics of the projects in the sample are summarized in Table IV.

On average, the software development organizations in the study had been in the software development business for about 7 years, although this varied from startup organizations to the oldest one at 47 years. In terms of their conceptual and process novelty, the projects were distributed according to the pattern

Table IV. A Summary of the Characteristics of the Projects in the Field Study

Project characteristic	Mean	Standard deviation
Project duration	11 months	9.6 months
Project team size	16 people	22.9 people
Percentage of total project hours spent on the design phase of the project	20.7%	8.47%
Vendor CMM level	2.6	1.07
Rework	15.7%	16.1%
Residual defects	25.16	61.09
Project cost overrun	16.40	16.43
Project conceptual novelty (4 level scale)	2.9	0.94
Project process novelty (4 level scale)	1.9	1.05

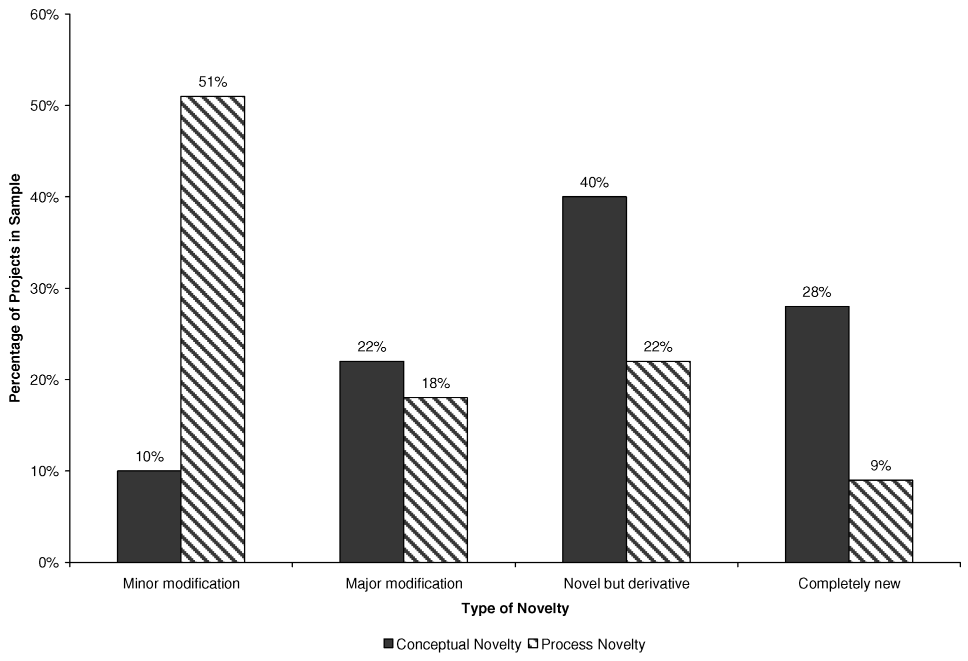

Fig. 3. Distribution of projects in terms of conceptual novelty and process novelty.

illustrated in Figure 3. The average level of conceptual novelty for these projects on a scale of one to four was approximately 2.9 (standard deviation 0.94) and process novelty 1.9 (standard deviation 1.05). Conceptual novelty was statistically significantly correlated with process novelty ($\beta = 0.353$, T-value $= 4.46$, statistically significant at the 99% confidence level). This suggests that conceptually novel projects are also more likely to use novel software development processes.

As the data show, approximately 40% of the projects in the sample were conceptually novel but based on a preexisting design while only 10% were truly

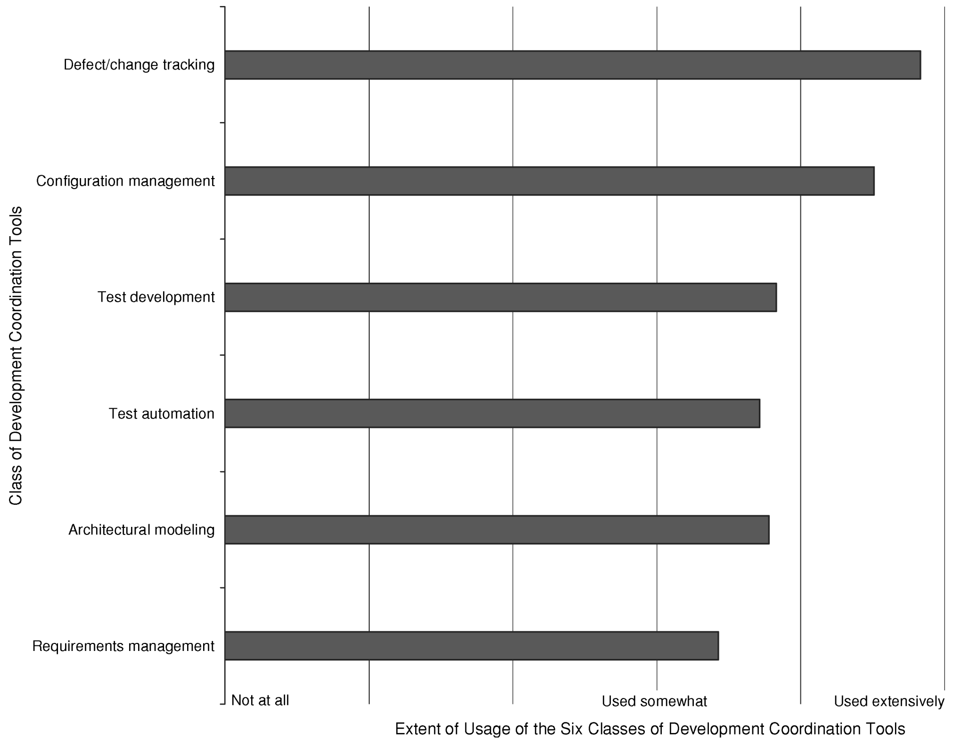

Fig. 4. Average usage of each class of development coordination tool in the sample.

routine. 28% were completely new. However, 51% of these projects involved no process novelty. The distribution pattern also shows that there was sufficient variance in the sample to rigorously assess the statistical interaction effects between novelty and development coordination tools [Baron and Kenny 1986]. The distribution pattern raises a cautionary note that relatively few projects (about 9%) were at the upper end of process novelty. However, the majority of projects (68%) in the study were relatively high on conceptual novelty. The ways in which this distribution pattern might influence our findings is discussed later in the article. Development coordination tool usage for various tool classes is summarized in Figure 4. All six classes of tools appear to be used but to varying degrees. Surprisingly, requirements management tools that have the most consistent performance impacts in the later results are relatively the least used. This gives some credence to recent alarms that the software industry underutilizes even tools that are already available [Spinellis 2005]. An alternative interpretation is that requirements management tools might have been used extensively primarily in the early project stages, leading the respondents to view its overall usage level as being lower relative to many other classes of tools.

Table V presents a correlation matrix for the data. The correlation matrix shows three noteworthy patterns in the correlations among various classes of development coordination tools. These patterns indicate which classes of

Table V. Correlation Matrix

Variable	Mean	SDev	Correlation matrix								
			1	2	3	4	5	6	7	8	9
1. Requirements managers	3.43	1.92	1.00								
2. Architectural modelers	3.78	1.87	0.46**	1.00							
3. Test automation tools	3.71	2.11	0.25**	0.42**	1.00						
4. Test case development tools	3.83	2.05	0.30**	0.41**	0.67**	1.00					
5. Configuration management tools	4.51	2.01	0.20*	0.13	0.32**	0.34**	1.00				
6. Defect & change request trackers	4.83	1.92	0.24**	0.31**	0.40**	0.26**	0.54**	1.00			
7. Development rework	15.6	16.03	0.17	0.00	−0.04	0.02	0.01	0.12	1.00		
8. Cost overrun	16.40	16.43	0.07	−0.04	−0.09	−0.13	0.15	0.08	0.37**	1.00	
9. Residual defects	25.16	61.08	0.00	0.22	0.23*	0.26*	0.00	0.12	0.25*	−0.02	1.00

* $p < 5\%$ ** $p < 1\%$; SDev refers to standard deviation.

tools were consistently used in conjunction with others and the correlations in the extent of their use in the projects in our sample. First, there is a statistically significant correlation between requirements management tools and all other classes of tools with the exception of configuration management tools. This suggests that requirements management tools were often used in conjunction with other tools. A second noteworthy pattern is the high, statistically significant correlation (0.67; $p < 0.01$) between test case development tools and test automation tools, which suggests a high likelihood that they the extent of use of one class of testing tools is directly correlated with the extent of use of the other. Third, these two classes of software testing tools have a moderate but significant correlation with defect/change request tracking tools and configuration management tools. This suggests that they are likely to be used together with testing tools, although to a lesser level.

5.2 Choice of Multiple Regression for the Statistical Analyses

The independent variables in the dataset are ordinal data measured on multilevel scales that capture the extent to which each class of tools was used in a project [Babbie 1973]. The dependent variables measuring software development performance are continuous. Furthermore, the dataset is relatively large by social science conventions and does not violate normal distribution assumptions. Assessing the relationships between various dimensions of performance and usage of the six classes of tools requires that the effect and statistical significance (i.e., assessing whether the relationships are real and not merely an artifact of chance) of the six predictors be simultaneously accounted for in the model [Hair et al. 1995]. In other words, analyzing the effect of one tool class at a time in the absence of others would lead to neglecting the possibility that the usage of one class of tools might deflate or inflate the relationship with performance for others that are not simultaneously included in a model [Aiken and West 1991]. Furthermore, any analytic technique that assumes Poisson distributions (e.g., log-linear regression) or dichotomous (e.g., logistic regression) or categorical variables (e.g., multinomial logistic regression) would be inappropriate. Therefore, given the type of data and the objective of the analyses, the most appropriate technique for statistically testing the relationships between tool class usage levels and the performance variables is multiple regression [Belsley et al. 1980; Cohen and Cohen 1983]. The choice of this regression-based analytical approach is consistent with a large body of empirical field studies using such data that have been conducted in the past three decades in social science fields such as MIS that have also examined software development teams and projects [Andres and Zmud 2002; Banker et al. 1998; e.g. Barki et al. 2001; Faraj and Sproull 2000; Ho et al. 2003; Kirsch et al. 2002]. In this approach, the usage levels of all six classes of tools are considered simultaneously in the regression runs. Researchers should be careful not to over interpret the results as implying that a single tool by itself will influence performance because the analyses rely on entering all six classes of tools in a single step. Therefore, the effects of one class of tools might be contingent on the presence of others in the model. This issue is discussed in further detail in discussing the interpretations of the statistical results.

5.3 Model Assessment Results

The three simultaneous regression equations mirroring (5), one each for rework, cost overrun, and residual defects, were statistically estimated using stepwise regression models using the data from all projects in the sample. The results of the analyses are summarized in Table VI. Each set of columns in Table VI corresponds to the three proxy variables measuring the efficiency and effectiveness dimensions of software development performance. The dataset did not exhibit any significant violations of the assumptions for multiple regression analyses and did not require any variable transformations. The series mean substitution algorithm was used for missing data points, as is conventional practice in such analyses. Both listwise and pairwise regression analysis approaches were used in the analyses. Each model was first estimated as a baseline model with all variables included, followed stepwise by interaction terms with conceptual novelty, process novelty, and finally multidimensional novelty. The results shown in Table VI are for the full model, which must be used for drawing interpretations [Aiken and West 1991].

Interpretation of the Statistical Results. The dependent variables in the three results columns of Table VI represent the three different dimensions of software development performance. The direction of the relationship between each class of tool and software development performance is indicated by the unstandardized regression coefficient, beta. The use of unstandardized beta coefficients is appropriate because the model is an explanatory model rather than a predictive model, where standardized betas would instead be appropriate [Pedhazur 1982]. Consider for example rework (Model A in Table VI). A positive (negative) beta sign for the rework column means that the use of that class of tools is associated with *increased (decreased)* rework. The other statistic shown in conjunction with each beta coefficient is a T-statistic. The T-statistic is a test for whether we have sufficient confidence in a statistical sense that the observed relationship is real and not by probabilistic chance. The minimal acceptable norm for T-values is to have a 95% (i.e., $p < 5\%$) or higher level of statistical confidence that the observed relationship is not by chance [Hair et al. 1995]. It is valid to draw conclusions only for beta coefficients with T-statistics with at least 95% statistical confidence. The T-statistic corresponding to a 95% confidence level is 1.69, 99% confidence is 2.36, and 99.9% confidence level is 3.17. The statistically significant relationships in Table VI are highlighted in **bold** type.

Only the results for development coordination tool classes in **bold** are statistically significant and should be used for drawing interpretations. The statistically-significant tool classes that are associated with *lower* rework, cost overrun, and residual defects have a negative sign and are highlighted in shaded box borders. These indicate improvements in software development performance. We used one-tailed T-tests and a statistical significance level of 5% (i.e., 95% statistical confidence level) to denote statistical significance. Our

Table VI. Impact of the Six Classes of Development Coordination Tools on Rework, Cost Overrun, and Residual Defects

Development Coordination Tool	Model A Rework	Model B Cost Overrun	Model C Residual Defects
	β(T-statistic)	**β(T-statistic)**	**β(T-statistic)**
STEP 1: Routine Projects (Baseline model)			
Constant (β_0)	(4.114)	(3.895)	(−.823)
Requirements management (β_{rmgr})	−2.444**(−2.806)	−4.587***(−5.110)	**2.308***(2.451)**
Architectural modeling ($\beta_{modeling}$)	−1.576(−2.699)	.827(1.374)	−.406(−.643)
Test automation (β_{test_auto})	4.887***(5.680)	5.894***(6.648)	−2.521**(−2.710)
Test case development (β_{test_case})	−4.243***(−5.985)	−5.448(−7.458)	1.343(1.753)
Configuration management (β_{config})	**2.600***(4.109)**	**1.808***(2.773)**	**.853(1.247)**
Defect & change request tracking ($\beta_{tracking}$)	.223(.375)	.764(1.246)	−1.363*(−2.119)
Conceptually Novel Projects STEP 2: Interaction terms for conceptual novelty			
CNew*Requirements management ($\beta_{rmgr\text{-}c}$)	**1.777**(3.012)**	**2.983***(4.908)**	−1.448**(−2.270)
Cnew*Architectural modeling ($\beta_{modeling-c}$)	**.999*(1.998)**	−.147(−.286)	**−.092(−.170)**
Cnew*Test automation (β_{test_auto-c})	−4.316***(−6.483)	−3.338***(−4.866)	**1.290*(1.792)**
Cnew*Test case development (β_{test_case-c})	**3.314***(6.075)**	**2.738***(4.872)**	−1.104*(−1.872)
Cnew*Configuration management ($\beta_{config-c}$)	−1.929***(−3.691)	.178(.330)	−.793(−1.403)
Cnew*Defect & change request tracking ($\beta_{tracking-c}$)	.152(.188)	−2.391**(−2.872)	**1.938*(2.219)**
Projects with Process Novelty STEP 3: Interaction terms for process novelty			
SPNew*Requirements management ($\beta_{rmgr\text{-}md}$)	**2.198**(2.396)**	**4.547***(4.811)**	−2.110*(−2.128)
SPNew*Architectural modeling ($\beta_{modelin-md}$)	1.340(1.329)	−2.585**(−2.488)	1.420(1.303)
SPNew*Test automation ($\beta_{test_auto\text{-}md}$)	−2.097*(−1.776)	−6.415***(−5.272)	**3.328**(2.607)**
SPNew*Test case development ($\beta_{test_case-md}$)	**2.151*(2.218)**	**5.777***(5.779)**	−.618(−.589)
SPNew*Configuration management ($\beta_{config-md}$)	−2.241*(−2.311)	−3.480***(−3.484)	−.647(−.617)

Table VI. *Continued.*

Development Coordination Tool	Model A Rework	Model B Cost Overrun	Model C Residual Defects
SPNew*Defect & change request tracking ($\beta_{tracking-md}$)	**−.694(−.620)**	**2.814******(2.441)**	−.313(−.259)
Projects with Multidimensional Novelty			
STEP 4: Interaction terms for multidimensional novelty			
MultiNew*Requirements management ($\beta_{rmgr\text{-}md}$)	−1.962*(−2.286)	−3.814***(−4.311)	1.181 (1.272)
MultiNew*Architectural modeling ($\beta_{modelin-md}$)	−.747(−.787)	**3.095******(3.167)**	−.645(−.629)
MultiNew*Test automation ($\beta_{test\_auto\text{-}md}$)	**1.990*(1.864)**	**5.627*******(5.113)**	−3.251**(−2.815)
MultiNew*Test case development ($\beta_{test\_case-md}$)	−1.694*(−1.974)	−5.085***(−5.750)	**1.874*(2.020)**
MultiNew*Configuration management ($\beta_{config-md}$)	**1.599*(2.323)**	**2.374******(3.348)**	.024(.033)
MultiNew*Defect & change request tracking ($\beta_{tracking-md}$)	.558(.646)	**−2.457*(−2.757)**	.217(.232)
Model R^2	**54.6%*****	**60.4%*****	**64.4%*****
Model F-value	**3.65*****	**3.81*****	**3.69*****

1. * 95%, ** 99%, *** 99.9% statistical confidence level; N = 209 projects.
2. Statistically significant relationships between tool class usage and performance are in **bold**.
3. Results in shaded boxes highlight tools that significantly are associated with higher development performance.
4. Negative signs indicate improvements and positive signs indicate decrease in software development performance.
5. The raw data used for the regression procedure is in the Appendix.
6. SPSS$^{(tm)}$ version 11.5 for Windows was used to compute the unstandardized beta, traditional R^2, and T-statistics.
7. SPSS 11.5's Base^{TM} version's implementation of the series mean substitution algorithm with pair wise and list wise regression procedures was used for estimation.

subsequent interpretations for some tool classes that—counterintuitively—were associated with decreased performance are also discussed, guided in part by our follow-up interviews with developers in the concluding phase of the study.

For drawing interpretations from the analyses, first consider just Model A (the first results column) in Table VI, which is the model for rework. In step 1 of the evaluation procedure, the six classes of development coordination tools are introduced in the model. This baseline model represents routine projects. Step 2 models the influence of each of the six classes of development coordination tools when projects are conceptually novel. Step 3 models the influence of each of the six classes of development coordination tools when projects involve process novelty. Step 4 models the influence of each of the six classes of development coordination tools when projects involve multidimensional novelty. We focus the later discussion primarily on the classes of tools that are associated positively with improvements in performance but also discuss possible reasons for the ones that are associated with decreases in performance (i.e., have a positive and statistically significant coefficient). The results in the columns for Models B and C in Table VI respectively correspond to cost overrun and residual defects, and must be interpreted as described for rework.

The routine project regression results in Table VI correspond to projects that do not involve novelty either in the underlying project concepts or in the development process. The corresponding results inform us about the relationships between performance and the usage level of the six classes of development coordination tools in such projects. The subsequent results for conceptual, process, and multidimensional novelty should be interpreted in a similar manner.

The amount of variance explained by each model, R^2 provides an indication of each model's goodness of fit. The model explained 54.6% of the variance in rework, 60.4% of the variance in cost overruns, and 64.4% of the variance in residual defects, suggesting that development coordination tool usage is a reasonable but imperfect predictor of software development performance. As can be expected, there was significant correlation between the three dependent variables. Specifically, cost overrun was significantly and positively correlated with development rework (correlation coefficient, $\gamma = 0.373$, $p < 0.001$) suggesting that higher levels of rework was associated with greater project cost overruns. It is important to note that cost overrun represents the additional cost that is passed on to the client and might not equal the actual cost incurred by the vendor. Likewise, rework was positively and significantly correlated with residual defects ($\gamma = 0.248$, $p < 0.027$), suggesting that greater residual defects were associated with more rework. Finally, residual defects did not have a statistically significant relationship with cost overruns, although the direction was negative ($\gamma = -0.017$, $p < 0.452$, nonsignificant). It is important to note that these relationships are correlations and do not establish a causal set of relationships. Establishing causality requires longitudinal data at different points in time for each project. However, the correlations between the three dependent variables does not confound the results because each dependent variable for each project type was analyzed as a separate regression model. We also tested whether the rework, cost overruns, and residual defects systematically varied depending on the presence of conceptual or process novelty. We found projects with process novelty are more likely to have defects remaining at the installation and warranty stages ($\gamma = 0.292$, T-value $= 2.072$, $p < 0.05$). These results are discussed in the next section.

5.4 Assessment of Threats to Validity

Before proceeding to a discussion of these results, it is important to consider our handling of five potential threats to the validity of the study. The first potential threat is *nonresponse bias*. Nonresponse bias refers to the potential bias that can result from the companies that chose to participate in the empirical study vis-à-vis those that did not participate [Babbie 1973]. To assess whether the non-responding organizations were biasing our sample, we made follow-up telephone calls to nine randomly selected nonresponding vendors (three in each country). Most nonparticipating organizations declined to participate for lack of time, concerns about sensitivity of the data, or because they were no longer in the custom software development business. Further, we also assessed differences between the early (first 50) and late (last 50) respondents and found no statistically significant differences in organizational characteristics such as company size and age. These checks suggest that nonresponse bias is not a persuasive threat to our findings.

The second potential threat is that is *key informant knowledge bias*, which refers to the ability of the responding managers to knowledgably provide an assessment of the various classes of tools used in the projects that were studied. We assessed this threat in the interviewing phase (phase 1) of the study through our interviews with project managers in 19 software development organizations. The lead project manager for each project was an appropriate respondent for each project because he/she is most likely to have good overall knowledge of the extent to which different classes of development coordination tools were used in a project [Weekley and Gier 1989]. In contrast, individual team members might have greater knowledge of the tools that they personally used in the project, but might not have accurate knowledge of the extent to which ones that they themselves did not use but other team members did. Ideally, we would want to collect information on the use of each class of tool from each team member for each project (and calculate an averaged team-level score). However, this approach would require collecting data directly from approximately 3,712 individuals (232*16 team members on average in each team) in 232 project teams, which was unfeasible given the limited access that we had to the participating organizations and our resource constraints. The client-side liaison managers from the client organization were the most appropriate source of information about deviation from budget because they were most likely to possess budget information as well as less likely to underreport it. In contrast, collecting this information directly from the software development organizations would have introduced a reporting bias [Shore et al. 1992]. Therefore, respondent knowledge bias is not a serious threat to the validity of this study.

The third potential threat is *the nature of the contractual agreement between the client and vendor companies*, that is, whether the contracts were fixed price or time-and-materials based [Gopal et al. 2003]. In our initial interviews, we found that software development organizations rarely use one or the other type of contracts; instead, they are usually a combination of fixed price elements coupled with time-and-materials components. Therefore, we did not specifically control for this distinction from a statistical standpoint.

The fourth potential threat is that of *construct validity*, that is, whether the measure for each construct actually measures what it was intended to measure. Two aspects of construct validity should be evaluated: (1) content and face validity and (2) level of granularity [Bagozzi et al. 1991; Lawther 1986; Paese and Switzer 1988]. We addressed the content validity threat by grounding the definitions of each class of tool in our interviews with actual software practitioners and extensively refining them in the field to ensure that they were interpreted unambiguously and clearly by the respondents in the larger-scale empirical survey phase that followed. The second aspect of construct validity is the granularity (i.e., aggregation across the entire project life cycle) at which tool class usage was measured. Recall that we measured the degree to which each class of tool was used over the life cycle of each project. An alternative would have been to provide a list of tools and collect data on each tool class's usage. However, this would have generated data that could not have been compared across projects because: (a) the terminology used to describe various tools varied widely among the two hundred plus software development organizations that we studied and many of the tools were proprietary (potentially causing unverifiable measurement errors across companies and projects) and (b) many integrated tool environments such as proprietary and commercial CASE tools provide different subsets of coordination functionality. Therefore, no cross-organizational analyses would have been possible across the projects in over two hundred different organizations without relying on a categorization of tools into broader generic functionality classes, grounded in our conceptual development interviews in Phases 1 and 2.

The final potential threat is that some tool classes such as test case development tools would be expected to be used largely or exclusively only in a few project phases. Tools that are typically used in fewer phases of typical projects might have evoked responses on the lower end of the measurement scale. We believe that this is not a serious threat to the validity of the study for two reasons. First, classes of tools that are typically used in fewer phases of software development projects would have generated a similar pattern of data, that is, the respondents evaluated tool class usage relative to what is typical in their view of the industry. Since regression analysis is based on variance in the data rather than absolute values, this would not systematically bias the results. The second piece of evidence to support this assessment appears in Figure 4, which shows that test case development, test automation, and architectural modeling tools, which are typically used in fewer phases of the software development process, are all around the middle point of the scale (i.e., neither too high nor too low, on average).

5.5 Limitations

This study has six limitations that warrant caution before generalizing the results. Unlike the threats to validity discussed in the preceding section, these are limitations of the dataset that cannot completely be ruled out. First, usage of various classes of development coordination tools in each of the projects analyzed in the study was based on the lead project manager's assessment. This was a necessary trade-off in order to collect a large dataset in a multinational setting from over 200 different companies in this study. It would be valuable to collect data that are more granular from each project team members in future studies. The model assumed that the impact of various classes of development coordination tools is linear. This assumption, while appropriate in this study, should be refined in future research. For example, future research can collect longitudinal data at multiple points in each project and examine whether some classes of tools (e.g., requirements managers) have curvilinear relationships with performance as a project progresses through different stages. Future work can also provide deeper insights by directly measuring the frequency and person-hours of use for each tool, which would be much more insightful than a simplistic perceptual assessment used in this study.

Second, the projects in the study were small to midsized projects. It is possible that the influence of development coordination tools will be different in large-scale projects. Since the complexity of the projects was not directly taken into consideration and there appears to be little variance in project size (a surrogate for complexity), caution is warranted in generalizing the results to complex projects. It is plausible that smaller projects could also be highly complex, requiring directly measuring and modeling project complexity in future studies of development coordination tools. In interpreting the results, it is also important to remember that the beta coefficients represent the effects of tool classes when all six classes are copresent in the model. In other words, drawing interpretations for one class of tools in isolation would be too simplistic and potentially misleading because each effect represents the performance effect from the usage of a given class of tools when the other five classes are also simultaneously accounted for in the model. Merely regressing one tool at a time would not have isolated the unique effects of a given class of tool either because some tools are used more in specific phases of the software development process and it would be almost impossible to ascribe causality to that class of tools unless the level of usage of all other classes of tools was identical across all projects being compared. It might, however, be possible to examine this issue in future research through a laboratory experiment in which only one tool class is varied across an experimental group and control group of projects. This approach will, however, introduce the trade-off that the study context would have to be a classroom setting rather than a real world industrial setting.

Third, residual defects were measured using recorded installation and warranty defects for completed projects. Not all existing warranty defects might have been detected at the time of the study. This is not a persuasive threat to our findings because if this is the case, the study underestimates rather than overestimates the effectiveness of various classes of development coordination tools. However, it is also possible that the undetected defects should not have been present in the first place, which is impossible to examine in the data. A related limitation is that we did not measure the severity of recorded defects, in essence treating all recorded defects as equal irrespective of their magnitude. This remains a promising refinement opportunity in future studies. Prototyping tools did not explicitly emerge in the preliminary interviews and prototyping functionality appears to have been subsumed under other classes of development coordination tools.

Fourth, the study was cross-sectional (i.e., collected at a single point in time) rather than longitudinal, which makes it impossible to statistically establish causality, that is, whether the use of a given tool class *causes* or is merely associated with improved performance. For the same reason, the analysis fails to account for learning curves in using development coordination tools; it is possible that some classes of tools did not generate observable benefits in projects with process novelty and multidimensional novelty because the development team had to go through a learning curve to benefit fully from their use. If the programmers do not become sufficiently familiar with a new tool, that tool is less likely to be effective in facilitating development performance improvements [Seeley 2003]. Therefore the long-term versus short-term impact of various classes of tools remains an important issue for future field studies; these must be conducted at the software development organization rather than project as the empirical unit of analysis. Another fruitful avenue for future research is to examine how the level of usage of various classes of tools varies with other characteristics of projects such as budget and duration.

Fifth, there is a wide variance in the characteristics of the projects and the organizations—such as project schedules, team size, design effort, vendor CMM level, national contexts, and age of the vendor—surveyed in this study. The risk that variations in those contextual variables might confound some effects of development coordination tools cannot be completely ruled out. However, there is also one commonality that lessens the severity of potential confounding effects: All of the projects were completed for US clients. One can reasonably presume that foreign software vendors who have completed projects for client companies in the same country might have sufficient commonalities in their characteristics that at least some of this threat would be mitigated. A second aspect that should also be considered is that the unit of analysis in our models is the project rather than the vendor organization; therefore, inclusion of other variables at different levels of analyses would not be methodologically permissible.

6. DISCUSSION

The overarching implication of the results is that the performance benefit of any class of development coordination tools is contingent on *the salient types of novelty* in a project. The majority of development coordination tools are associated with some dimension of higher performance but the dimension of performance that they enhance—efficiency (e.g., reducing rework, and preventing cost overruns) or effectiveness (e.g., reducing software defects)—varies systematically based on whether a project involves conceptual novelty, process novelty, neither, or multidimensional novelty (both conceptual and process novelty). The results provide nuanced insights—some counter to prevailing beliefs—into the portfolio of development coordination tool classes associated with higher software development performance in various types of projects.

The overall pattern of results for the influence of each class of development coordination tool on software development performance is summarized in Table VII. (These patterns are based on the statistical results in Table VI.) Here, rework reduction and prevention of cost overruns represent proxies for the efficiency dimension of software development performance and reduction in post-delivery defects as the effectiveness dimension.

Three overarching patterns in Table VII are noteworthy. First, the use of many classes of tools in any type of project is associated with increases in some dimensions of performance but decreases in others. It appears that software development efficiency is achieved through the use of some classes of tools at the expense of software development effectiveness and vice versa. This can be viewed as an *efficiency-effectiveness trade-off*. Consider requirements management tools as one example. In routine projects, such tools are positively associated with efficiency but negatively with effectiveness. In projects with only one type of novelty, such tools are associated with improvements in effectiveness but decreases in efficiency. Test case development tools and test automation tools also reveal similar patterns. Second, architectural modeling tools and configuration management tools are exceptions to this in that they do not introduce an efficiency-effectiveness tension in any type of project. Third, classes of development coordination tools that are associated with higher performance in multidimensionally novel projects—counter to the intuitive assumption—are not a perfect superset of those that are associated with higher performance in projects with conceptual and process novelty.

We discuss next the findings related to each class of development coordination tools for projects with no novelty, process novelty, conceptual novelty, and multidimensional novelty. Our interpretations are based on the sign of the path coefficients (i.e., beta values, which indicate the direction of influence of each class of development coordination tool on each of the three performance variables) as well as its statistical confidence (significance) level in Table VI. Caution should be exercised in interpreting these results as being causal rather than associative; the former requires longitudinal data over multiple periods for each project and ruling out of alternative hypotheses [Duncan 1985]. Causality is implicitly assumed but it cannot be statistically tested with our cross-sectional data.

The following discussion references beta values, that is, the sign of the path coefficients in the regression model that describe the direction of the relationships between tool class usage and software development performance. The letters in the beta subscripts in Table VI and the following discussion denote the class of tools and the type of project that the beta value represents separated by a dash. For example, $\beta_{rmgr\text{-}r}$ represents the beta coefficient for requirements management tools for routine projects, matching the beta labels used in Table VI.

6.1 Requirements Management Tools

Requirements management tools were significantly associated with higher software development performance along at least one of the three dimensions of

Table VII. The Relative Influence of Various Classes of Development Coordination Tools

Class of development coordination tools	Project Novelty Type											
	Routine			Conceptual			Process			Multidimensional		
	Rework	Cost Overrun	Residual Defects	Rework	Cost Overrun	Residual Defects	Rework	Cost Overrun	Residual Defects	Rework	Cost Overrun	Residual Defects
Requirements managers	▲	▲	▽	▽	▽	▲	▽	▽	▲	▲	▲	
Architectural modelers	▲			▽				▲			▽	
Test automation tools	▽	▽	▲	▲	▲	▽	▲	▲	▽	▽	▽	▲
Test case development tools	▲	▲		▽	▽	▲	▽	▽		▲	▲	▽
Configuration management tools	▽	▽		▲			▲	▲		▽	▽	
Defect and change request trackers			▲		▲	▽		▽			▲	

Notes: Only the statistically significant results are summarized in the table.
▲indicates that the tool was associated with higher software development performance.
▽indicates that the tool was associated with lower software development performance.

performance across all four types of projects. However, the nature of their impacts varied across project types: Their use was associated with lower rework and cost overrun in routine and multidimensionally novel projects (as indicated by the negative and statistically significant $\beta_{rmgr\text{-}r}$ and $\beta_{rmgr\text{-}md}$ in Table VI), but with fewer residual defects in projects with only a one type of novelty ($\beta_{rmgr\text{-}sp}$ and $\beta_{rmgr\text{-}c}$ in Table VI). Conceptually, this implies that the capability to maintain requirements and interdependencies between requirements provided by requirements management tools enhances development *efficiency* in both routine and multidimensionally novel projects. In projects with only one form of novelty (i.e., either conceptual or process novelty), requirements management tools enhance the effectiveness dimension of performance. However, their use appears to be associated with increased cost overrun and rework in such projects, suggesting that cost overruns might be associated with greater defect detection and their subsequent elimination during the development process. Another interpretation for this pattern of results is that requirements management tools help maintain interdependencies among interacting requirements in singularly novel projects, which lowers residual defects but that a direct cost overhead might be associated with the comprehensive and formalized requirements management process imposed by their use.

6.2 Architectural Modeling Tools

Architectural modeling tools were associated with significantly lower development rework in routine projects (suggested by a negative and statistically significant $\beta_{modeling-r}$ term in Table VI). This finding is consistent with the logic in Section 2 that a good high-level design and a sound architecture that such development coordination tools facilitate enhances software development efficiency when requirements are relatively well definable and the envisioned outcome can be more completely specified at the outset of the project. Architectural modeling tools are associated with a decrease in cost overruns in projects with process novelty. An interpretation for this is that by using these development coordination tools to coordinate the high-level design/architecture of the software frees up development resources to execute the relatively novel process that implementing it entails, in turn enhancing software development efficiency. We found no statistical evidence that such tools enhanced either software development efficiency or effectiveness in conceptually novel or multidimensionally novel projects. One interpretation for this is that in conceptually novel projects, architectural modeling functionalities do not have a significant influence because the needs that the project must fulfill are incompletely known at the outset. Instead, a more evolutionary approach is required to specify iteratively the high-level design of the system. In projects with multidimensional novelty, tools that facilitate high-level architecture might impose preexisting design patterns that they natively support and might lead the development team to make design assumptions that follow from preexisting development processes but do not necessarily correspond to the new concepts that the project is intended to implement. It is impossible to deduce from the available data whether the higher number of residual defects found would have been missed without the use of such development coordination tools. This issue is critical for understanding whether architectural modeling tools ought to be used in novel projects deserves further examination in future research. Future research is also needed to examine whether there are other benefits (such as client satisfaction with the delivered system) that arise from their use, especially in complex projects.

6.3 Test Automation Tools

Test automation tools were associated with a higher level of at least some dimension of software development performance across all four types of projects, although they also introduced efficiency-effectiveness tradeoffs in three of the four types of projects. In both routine and multidimensionally novel projects, their use was associated with fewer residual defects (indicated by the negative and statistically significant $\beta_{test\_auto\text{-}r}$ and $\beta_{test\_auto\text{-}md}$ in Table VI). This suggests that in routine projects, automated execution of test cases lowers the incidence of defects. Surprisingly, their use also is associated with increased rework and cost overruns in such projects. An interpretation for this is that the additional cost and rework might result from the increase in defects that they help *detect* during the development process and that need to be fixed. It is also possible that the additional cost of test automation tools will arise from the learning curve that the developers might have to go through before fully benefiting from their use. The cross-sectional nature of our data, however, constrained our ability to assess empirically this possibility. In projects with either conceptual or process novelty, the use of test automation tools was associated with both lower rework and lower cost overruns (indicated by negative and statistically significant $\beta_{test\_auto\text{-}c}$ and $\beta_{test\_auto\text{-}sp}$ in Table VI). One interpretation for this finding is that automation of test case execution in the presence of a singular novelty frees up developers' attention from rote testing activities to the more creative aspects such as the novel conceptual or process framework, thereby increasing the efficiency dimension of software development performance. Surprisingly, their use was associated with greater residual defects in projects with process novelty. An interpretation for this is that automated testing might provide more frequent opportunities for detecting defects in the simultaneous presence of novel software development processes. A similar pattern for conceptually novel projects might further suggest that automation of testing activities can plausibly increase the incidence of defects when the novelty of project concepts hinders the ex ante formal specification of project details.

6.4 Test Case Development Tools

Test case development tools help create system-level test-case descriptions that confirm whether project requirements are met. Test case development tools therefore serve a validation function that ensures that the system level tests can be mapped back to the *known* requirements, thus reducing later rework and the costs associated with it. Their use was associated with higher development performance along at least one dimension in all projects except projects with process novelty. They enhanced efficiency in routine (see $\beta_{test\_case\text{-}r}$ in

Table VI) and multidimensionally novel projects, and effectiveness in projects with conceptual novelty ($\beta_{test\_case\text{-}c}$ in Table VI). Overall, the performance benefits of test case development coordination tools appear to be most pronounced in routine and multidimensionally novel projects, with somewhat less pronounced benefits in conceptually novel projects. An interpretation for this difference is that effective use of such development coordination tools requires that the requirements be relatively well known and relatively stable from the outset, which is less likely in conceptually novel projects. In the case of multidimensionally novel projects, they perhaps help cope with the sheer complexity of developing test cases in the simultaneous presence of conceptual and process novelty. However, test case development tools introduce the aforementioned efficiency-effectiveness trade-off in all types of projects except routine projects; that is, they lower some dimension of performance while increasing others. They lower efficiency in projects with singular novelty (see $\beta_{test\_case\text{-}c}$ and $\beta_{test\_case\text{-}sp}$) and lower effectiveness in projects with multidimensional novelty (see $\beta_{test\_case\text{-}md}$). An interpretation for the former is that the absence of precise upfront project requirements makes their use less efficient in conceptually novel projects and mapping the known requirements to test cases in projects with process novelty is more time consuming because of the absence of a preestablished development process. Another plausible interpretation is that such tools assume an established software process (which is not the case if the software process itself is novel), leading to critical errors in the system going undetected while executing the resulting test cases. Correcting these defects later in the process then imposes additional costs and rework. A competing interpretation is that additional rework and costs arise from the correction of defects that such development coordination tools help identify. However, this interpretation is less plausible because the use of such tools were not significantly associated with fewer residual defects. An interpretation for the tradeoffs observed for multidimensionally novel projects is that although such tools facilitate efficient development of test cases, this efficiency comes at the cost of comprehensiveness and precision in developing them.

6.5 Configuration Management Tools

Configuration management tools facilitate keeping track of associations among various versions of all of the project's artifacts throughout the development life cycle. The use of configuration management tools was associated with higher development performance only in projects with a singular type of novelty ($\beta_{config\text{-}c}$ and $\beta_{config\text{-}sp}$). No efficiency-effectiveness trade-offs were observed in such projects. The capability to associate versions of artifacts with builds of the system throughout the development process helps maintain consistency with evolving requirements that emerge as development progresses in conceptually novel projects. This in turn proactively mitigates potential mismatches between the most recent versions of requirements, specifications, and system artifacts associated with them, thereby lowering the need for additional rework to correct the ensuing defects. Projects involving novel development processes also benefit from the use of configuration management tools because they help coordinate the mappings among requirements, specifications, and system artifacts relatively independently of the development process. This increases the efficiency with which traceability information can be maintained, reducing the extent of cost overruns and additional rework. Although novel projects appear to benefit from the use of configuration management tools, their use is associated with lower software development efficiency in routine and multidimensionally novel projects. An interpretation for the former finding is that keeping track of various versions of artifacts imposes an unnecessary overhead when requirements are stable and might have fewer versions relative to singularly novel projects. An interpretation for the latter is that it is too complex and time consuming to keep track of requirements using such tools when requirements have too many versions because of evolving requirements and the simultaneous absence of an established software development process.

6.6 Defect and Change Request Tracking Tools

Defect and change request tracking tools were associated with higher software development effectiveness in routine projects and with higher software development efficiency in multidimensionally novel projects, without any efficiency-effectiveness trade-offs. In conceptually novel projects, they were associated with higher software development efficiency but also with a greater incidence of residual defects. An interpretation for this finding is that in conceptually novel projects, where project requirements are incompletely known at the outset, defect & change request tracking tools facilitate rigorous tracking of project requirements as they evolve or change, plausibly increasing the likelihood of detecting more defects in the design of the software. Our interviews also suggested an additional interpretation for why more residual defects might exist: The functionality to enforce assignment of authority for incorporating various types of changes to the system puts certain changes out of the scope of some team members' work authority and access rights/ privileges. Our interviews also suggested that at times a defect is observed by a team member in one of the stakeholder organizations but the inability to report it in a timely manner due to the authority allocation rigor enforced by such tools can lead to its going unrecorded until much later in the development process. This constraint in turn leads to defects being detected yet remaining in the system, which in turn can increase the incidence of defects at the time of delivery. It is also noteworthy that the use of defect and change request tools was skewed towards higher end of the scale. An interpretation for this is that in the outsourcing context in which the study was conducted, software development organizations are likely to use these development coordination tools more for two additional reasons: (1) they serve a bookkeeping function for recording defects for CMM-type certifications and (2) they might create an improved perception of quality management practices among clients. In projects with process novelty, the use of such tools is associated with increased cost overrun. An interpretation for this result is that they provide an elaborate mechanism for recording defects and change requests but the absence of a well-established process hinders the

Tool Class	Project Novelty Type			
	Routine	Conceptual	Process	Multidimensional
Requirements managers	○	○	○	●
Architectural modelers	●		●	
Test automation tools	○	○	○	○
Test case development tools	●	○		○
Configuration management tools		●	●	
Defect & change request trackers	●	○		●

Notes: ○ denotes performance benefits with efficiency-effectiveness tradeoffs; ● benefits without tradeoffs.

Fig. 5. Overall patterns of the performance implications of the various classes of development coordination tools.

efficient incorporation of such requests into the development process, in turn lowering software development efficiency.

6.7 Summary and Implications for Software Development Practice

The most important insight for practice from these results is that all classes of development coordination tools except architectural modeling tools and configuration management tools introduce efficiency-effectiveness trade-offs in some types of projects. Consistent with what has been suspected [Seeley 2003], a mismatch between the choice of tools and type of project can therefore *worsen* some dimension of software development performance even when it is associated with improvements other dimensions. This is what we labeled as the *efficiency-effectiveness trade-off*. Figure 5 synthesizes the pattern of results from the statistical analyses to illustrate the trade-offs across projects involving different combinations of novelty.

To summarize, classes of development coordination tools that are associated with improved software development performance *without trade-offs* in specific types of projects are the following.

—*In routine projects*, architectural modeling, test case development, and defect/change request tools are associated with higher software development performance.

—*In conceptually novel projects*, configuration management tools are associated with higher software development performance.

—*In projects using novel software processes*, architectural modeling and configuration management tools are associated with higher software development performance.

—*In multidimensionally novel projects*, requirements management and defect/change request tools are associated with higher software development performance. The classes of development coordination tools that are associated with higher performance under multidimensional novelty are therefore not a true superset of conceptual and process novelty, as commonly asserted in practice.

Software development organizations therefore must first carefully consider the characteristics of each individual project and then select a set of development coordination tools are more likely to have the greatest impact on its development process.

7. CONCLUSIONS

As both the complexity and novelty of software projects continue to increase, a growing number of software development organizations are adopting development coordination tools in an attempt to improve development performance and productivity. In the absence of any prior field research on when and how such tools improve software development performance, many organizations have leaned towards adopting a broad array of different tools under the presumption that they can only enhance software development performance. Others have avoided extensively adopting such tools because some industry observers have cautioned that it might be wiser to not use a tool than to use the wrong tool [Brown 2003; Spinellis 2005]. The objective of this early field study was to assess empirically the relationship between the use of various classes of development coordination tools and software development performance in the actual industrial context in which they are used.

We first identified and conceptually delineated six distinct classes of software development coordination tools used in the industry based on in-depth interviews in 19 software development organizations. We then used the results of this phase to assess empirically the relationships between the use of these tool classes and software development performance using primary field-based data collected from 209 different software development companies. We statistically analyzed this data through regression analysis techniques.

Overall, the results provide new insights into the relationship between the use of various classes of development coordination tools and software development performance. The results reveal two broad patterns. First, the dimension of performance that each class of tool impacts—that is, whether it mitigates development rework, controls budget slippage, or incidence of post-delivery software defects—depends on whether the project involves conceptual, process, or multidimensional novelty. A given class of development coordination tools that is beneficial in one type of project can have no impact—or worse, a negative impact—in another. Second, the use of many classes of tools introduces efficiency-effectiveness tradeoffs; that is, it can enhance one dimension of performance while lowering another. We found two exceptions—architectural modeling tools and defect/change request-tracking tools—that do not impose such tradeoffs. Although this study provides an initial step, considerable potential exists for further fieldwork that can expand our understanding about how development coordination tools enhance software development performance.

APPENDIX: RAW DATA

Usage of six classes of development coordination tools 1 = “not at all,” 4 = “somewhat,” and 7 = “to a great extent.”						Dimensions of Software Development Performance		
T1	T2	T3	T4	T5	T6	P1	P2	P3
1	1	1	1	1	7	16	10	40
6	4	1	6	7	1	30	15	8
3	4	4	4	5	5	16	16	25
1	1	1	2	1	1	16	16	25
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
5	5	5	5	4	4	0	16	25
3	4	4	4	5	5	16	16	25
6	2	1	1	6	6	50	5	2
3	4	4	4	5	5	16	16	25
7	5	7	7	5	5	16	16	25
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
4	4	4	4	4	4	16	16	25
1	4	1	1	1	1	16	16	10
1	1	1	1	6	7	30	10	10
1	1	1	1	1	5	16	30	25
3	4	4	4	5	5	16	16	25
1	1	7	1	1	4	50	80	1
4	4	4	4	4	4	16	0	25
3	3	4	4	4	4	16	20	20
3	5	2	2	4	5	16	16	2
1	1	5	5	7	7	16	16	25
3	3	3	3	3	3	16	16	6
3	4	4	4	5	5	16	16	25
1	7	6	6	2	5	5	40	350
1	4	1	1	1	5	5	0	25
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
5	5	2	4	3	4	40	50	2
1	1	3	3	1	2	16	16	25
5	5	5	5	5	5	16	1	1
1	1	1	1	1	1	10	16	5
1	1	1	1	1	1	5	20	12
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
1	1	1	3	4	3	16	20	4
2	3	4	5	5	5	0	16	25
6	2	6	6	2	7	0	5	25
2	5	6	6	7	7	16	4	1
4	5	6	4	2	7	16	16	25
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
5	1	1	6	6	6	16	10	25
2	4	4	3	5	4	16	10	25

Usage of six classes of development coordination tools 1 = “not at all,” 4 = “somewhat,” and 7 = “to a great extent.”						Dimensions of Software Development Performance		
3	1	2	2	5	3	20	25	4
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
1	1	7	7	7	7	10	16	25
1	5	7	7	7	7	16	2	25
3	4	4	4	5	5	16	16	25
6	6	3	5	7	7	15	30	25
5	6	7	7	6	7	16	20	25
5	5	3	3	3	5	10	35	11
1	7	1	1	1	5	16	3	8
2	2	3	3	4	5	110	5	25
2	1	1	1	4	1	20	20	7
1	1	1	2	3	1	16	16	5
1	3	1	1	5	5	16	16	2
3	3	3	4	5	4	30	2	40
6	6	3	3	5	6	50	60	20
1	3	1	1	3	1	25	5	11
1	6	1	1	1	6	16	16	25
5	6	6	7	6	6	20	40	400
1	1	3	3	1	1	2	0	4
6	6	4	5	5	6	16	30	25
2	4	4	4	3	3	16	16	25
7	7	6	6	6	6	16	16	7
5	7	7	7	7	7	16	16	25
1	1	1	1	1	6	10	1	30
3	4	6	6	6	5	50	50	100
1	1	7	7	4	4	16	10	25
5	2	6	3	4	3	16	30	35
3	4	4	4	5	5	16	16	25
5	5	4	3	7	7	50	50	25
4	5	4	5	5	1	16	16	25
4	5	5	4	4	5	16	16	25
3	6	2	2	6	6	15	10	30
7	7	5	5	3	6	16	16	2
3	4	4	4	5	5	16	16	25
7	5	5	5	6	7	20	35	12
6	6	6	6	6	6	16	1	5
2	3	3	5	7	5	10	10	7
4	4	3	3	3	2	10	2	25
3	5	3	2	5	5	16	10	10
2	2	1	1	1	1	12	15	25
6	6	6	6	6	6	10	10	25
4	5	5	4	4	6	16	30	8
4	4	4	4	4	4	16	20	25
2	6	6	7	4	7	16	5	25
6	3	3	3	4	4	16	16	25
1	4	3	1	1	6	0	5	25
7	5	7	4	5	7	13	5	30

Usage of six classes of development coordination tools 1 = "not at all," 4 = "somewhat," and 7 = "to a great extent."						Dimensions of Software Development Performance		
3	5	7	5	7	7	5	10	4
3	6	2	2	6	6	15	10	30
4	4	5	4	5	4	20	10	7
2	5	1	1	7	4	30	10	5
4	3	3	4	1	4	0	3	25
4	4	4	4	4	4	0	25	25
1	4	1	6	1	3	20	10	25
1	4	1	7	1	1	0	10	3
6	7	4	4	5	7	16	16	25
2	6	6	5	6	6	15	20	15
7	6	1	3	5	6	25	15	8
5	5	5	5	4	4	16	16	25
1	1	2	1	7	6	20	16	0
3	4	4	4	5	5	16	16	25
5	4	5	5	4	5	10	15	25
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
5	1	1	4	4	6	25	50	15
3	4	4	4	5	5	16	16	25
4	4	6	6	7	4	20	10	25
5	2	5	4	7	7	16	16	25
6	5	2	6	6	7	30	20	35
3	3	4	7	6	6	10	25	25
5	4	6	2	5	5	16	16	25
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
1	3	1	1	7	7	20	20	2
2	5	4	2	6	7	15	20	18
3	4	4	4	5	5	16	16	25
3	5	7	7	7	5	16	16	25
3	4	2	2	1	1	16	16	25
1	2	1	2	1	2	10	16	25
1	3	3	4	6	6	10	15	7
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
5	4	1	2	3	4	15	20	10
4	5	5	5	6	5	10	2	5
1	4	7	7	7	7	16	16	25
1	1	1	1	7	1	20	16	22
6	6	6	6	6	6	15	5	25
4	4	4	1	4	7	16	16	25
3	4	4	4	5	5	16	16	25
5	5	5	5	6	6	16	10	25
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
5	3	5	5	5	5	15	16	25
5	5	1	7	5	7	16	80	7

Usage of six classes of development coordination tools 1 = "not at all," 4 = "somewhat," and 7 = "to a great extent."						Dimensions of Software Development Performance		
3	3	7	7	7	7	0	0	25
6	6	6	6	6	6	16	16	12
4	4	3	4	3	6	16	16	25
1	1	1	1	5	5	16	16	25
3	4	4	4	5	5	16	16	25
4	4	4	4	4	4	20	0	25
3	6	6	6	2	2	0	10	25
5	5	7	6	5	7	10	15	25
4	4	4	4	4	4	16	16	25
1	1	7	7	7	7	5	15	25
1	1	1	1	7	6	16	16	25
4	5	5	4	5	5	0	15	29
3	3	4	2	7	7	16	16	25
3	4	4	4	5	5	16	16	25
1	2	3	1	6	5	10	5	25
5	5	6	5	5	5	16	16	25
6	1	2	2	7	7	6	40	3
5	5	6	6	6	5	15	10	75
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
4	4	7	7	7	7	16	16	15
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
5	7	7	7	1	6	6	10	8
4	7	7	4	7	7	10	12	25
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
1	1	5	5	5	5	20	16	25
3	5	5	6	6	5	30	8	21
4	5	5	5	5	5	16	16	25
5	5	4	6	5	6	16	16	25
2	5	5	5	5	5	20	10	5
5	1	1	1	5	1	10	16	13
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
5	5	2	6	5	5	10	10	25
3	4	4	4	5	5	16	16	25
6	6	1	1	5	7	10	12	25
5	1	1	1	1	1	16	16	25
3	3	4	4	6	6	5	16	5
5	5	5	4	4	4	16	16	25
5	5	1	5	5	5	16	16	25
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
5	7	6	5	2	3	16	20	32
3	4	4	4	5	5	16	16	25
1	5	5	5	1	1	20	20	25
3	2	1	2	6	7	16	60	100
4	1	4	1	6	7	16	0	25

Usage of six classes of development coordination tools 1 = “not at all,” 4 = “somewhat,” and 7 = “to a great extent.”						Dimensions of Software Development Performance		
4	3	5	3	4	4	12	15	2
3	4	4	4	5	5	16	16	25
3	4	4	4	5	5	16	16	25
1	1	1	7	7	1	0	10	25
3	4	4	4	5	5	16	16	25
1	5	5	3	6	5	5	2	25
5	5	2	1	5	5	5	5	40
7	4	2	2	2	2	5	16	25
2	2	2	2	2	2	3	10	0
3	5	5	4	6	6	10	16	25
3	4	4	4	5	5	16	16	25
6	3	7	7	7	7	16	15	25
1	5	5	4	6	6	5	0	18
1	1	3	1	7	7	16	16	25
3	4	4	4	5	5	16	16	25
4	1	2	1	1	3	10	5	25
5	2	1	2	4	4	3	1	4
3	4	4	4	5	5	16	16	25

1. Tool usage variables legend: T1 = Requirements management; T2 = Architectural modeling; T3 = Test automation; T4 = Test case development; T5 = Configuration management; T6 = Defect & change request tracking.
2. Software development performance variables legend: P1 = Cost overrun as % of planned budget; P2 = Rework as % of total project function points; P3 = Residual defects in installation and warranty stages.
3. Usage of each class of development coordination was measured using the project manager's perceptual evaluation.
4. Mean substitution was used for missing data in the analyses. See Table 5 for series means.
5. Defect data represents only the recorded defects and was self-reported by the project manager for each project.
6. Scales/ measurement items in the questionnaire are described in Section 4.1.

ACKNOWLEDGMENTS

Comments and feedback from Ashley Bush and four anonymous reviewers are gratefully acknowledged.

REFERENCES

Abdel-Hamid, T., Sengupta, K., and Ronan, D. 1993. Software project control: An experimental investigation of judgment with fallible information. *IEEE Trans. Soft. Engin. 19*, 6, 603–612.

Adelson, B. and Soloway, E. 1985. The role of domain experience in software design. *IEEE Trans. Soft. Engin. 11*, 11, 1351–1360.

Adler, P. S. 1995. Interdepartmental interdependence and coordination—the case of the design/manufacturing interface. *Organiz. Sci. 6*, 2, 147–167.

Aiken, L. and West, S. 1991. *Multiple Regression: Testing and Interpreting Interactions*. Sage, Newbury, CA.

Andres, H. P. and Zmud, R. W. 2002. A contingency approach to software project coordination. *J. Manag. Info. Sys. 18*, 3, 41–70.

Babbie, E. R. 1973. *Survey Research Methods*. Wadsworth, Belmont, CA.

Bagozzi, R. P., Yi, Y., and Phillips, L.W. 1991. Assessing construct validity in organizational research. *Admin. Sci. Quarterly. 36*, 3, 421–458.

Banker, R., Davis, G., and Slaughter, S. 1998. Software development practices, software complexity, and software maintenance performance. *Manag. Sci. 44*, 4, 433–450.

BARKI, H., RIVARD, S., AND TALBOT, J. 2001. An integrative contingency model of software project risk management. *J. Manag. Info. Syst. 17*, 4, 37–69.

BARON, R. AND KENNY, D. 1986. The moderator-mediator variable distinction in social psychological research: Conceptual, strategic, and statistical considerations. *J. Person. Soc. Psychol. 51*, 1173–1182.

BASKERVILLE, R. AND PRIES-HEJE, J. 1999. Grounded action research: A method for understanding IT in practice. *Account. Manag. Info. Techno. 9*, 1, 1–23.

BELSLEY, D.A., KUH, E., AND WELSCH, R.E. 1980. *Regression Diagnostics: Identifying Influential Data and Forces of Collinearity*. Wiley, New York.

BROWN, D. 2003. The developer's art today: Aikido or sumo? *ACM Queue 1*, 6.

COHEN, J. AND COHEN, P. 1983. *Applied Multiple Regression Analysis for the Behavioral Sciences*. Lawrence Erlbaum, Hillsdale.

CRESWELL, J. 1994. *Research Design: Qualitative and Quantitative Approaches*. Sage Publications, Newbury, CA.

CRONBACH, L.J. AND MEEHL, P.E. 1955. Construct validity in psychological tests. *Psychol. Bull. 52*, 4, 281–302.

DUNCAN, O. 1985. Path analysis: Sociological examples. In *Causal Models in Social Sciences*, H. Blalock (Ed.). Aldine Publishing, Hawthorne, NY,

FARAJ, S. AND SPROULL, L. 2000. Coordinating expertise in software development teams. *Manag. Sci. 46*, 12, 1554–1568.

GLASER, B. AND STRAUSS, A. 1967. *The discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing Company, Chicago.

GOPAL, A., SIVARAMAKRISHNAN, K. KRISHNAN, M., AND MUKHOPADHYAY, T. 2003. Contracts in offshore software development: An empirical analysis. *Manag. Sci. 49*, 12, 1671–1683.

GOTTESDIENER, E. 2003. Requirements by collaboration. *IEEE Soft.* March–April, 52–55.

GROTH, R. 2004. Is the software industry's productivity declining? *IEEE Soft. 21*, 6, 92–94.

GUINAN, P., COOPRIDER, J., AND SAWYER, S. 1997. The effective use of automated application development tools. *IBM Syst. J. 36*, 1, 124–139.

HAIR, J.F., JR., ANDERSON, R.E. TATHAM, R.L., AND BLACK, W.C. 1995. *Multivariate Data Analysis*. Prentice Hall, Englewood Cliffs, NJ.

HO, V., ANG, S., AND STRAUB, D. 2003. When subordinates become IT contractors: Persistent managerial expectations in IT outsourcing. *Inform. Syst. Resear. 14*, 1, 66–86.

KING, S. AND GALLIERS, R. 1994. Modelling the case process: Empirical issues and future directions. *Info. Softw. Tech. 36*, 10, 587–596.

KIRSCH, L., SAMBAMURTHY, V. KO, D., AND PURVIS, R. 2002. Controlling information systems development projects: The view from the client. *Manag. Sci. 48*, 4, 484–498.

KOKOL, P. 1989. Formalization of the information system development process using metamodels. *ACM SIGSOFT Softw. Engin. Notes 14*, 5, 118–122.

KUMAR, N., STERN, L.W., AND ANDERSON, J.C. 1993. Conducting interorganizational research using key informants. *Academy Manag. J. 36*, 6, 1633–1651.

LAWTHER, W.C. 1986. Content validation: Conceptual and methodological issues. *Rev. Public Person. Admini 6*, 3, 37–49.

LEE, A.S. A scientific methodology for MIS case studies, *MIS Quart.*, 1989, 33–50.

MACCORMACK, A., VERGANTI, R., AND IANSITI, M. 2001. Developing products on internet time: The anatomy of a flexible development process. *Manag. Sci. 47*, 1, 133–150.

MCAFEE, A. 2003. When too much IT knowledge is a dangerous thing. *MIT Sloan Manag. Rev. 44*, 2, 83–89.

MESSERSCHMITT, D. AND SZYPERSKI, C. 2003. *Software Ecosystem*. MIT Press, Cambridge, MA.

MINGERS, J., 2001. Combining IS research methods: Towards a pluralist methodology. *Info. Sys. Resea. 12*, 3, 240–259.

MOOKERJEE, V.S. AND CHIANG, R. 2002. A dynamic coordination policy for software system construction. *IEEE Trans. Softw. Engin. 28*, 6, 684–694.

MORGAN, G. AND SMIRCICH, L., SAWYER, S. 1980. The case for qualitative research. *Acad. Manag. Rev. 5*, 4, 491–500.

Mumford, M., Costanza, D., and Connelly, M. 1996. Item generation procedures and background data scales: Implications for construct and criterion-related validity. *Person. Psychol. 49*, 2, 361–398.

Orlikowski, W. 1989. Division among the ranks: The social implications of case tools for system developers,. In *Proceedings of the Tenth International Conference on Information Systems*, J. Degross, J. Henderson and B. Konsynski Eds. ACM Press.

Orlikowski, W. J. 1993. Case tools as organizational change: Investigating incremental and radical changes in systems development. *MIS Quart. 17*, 309–340.

Paese, P.W. and Switzer, F.S., III. 1988. Validity generalization and hypothetical reliability distributions: A test of the schmidt-hunter procedure. *J. Appl. Psychol. 73*, 2, 267–274.

Pedhazur, E.J. 1982. *Multiple Regression in Behavioral Research: Explanation and Prediction*. Dryden Press, New York, NY.

Ramesh, B. 1998. Factors influencing requirements traceability practice. *Comm. ACM 41*, 12, 37–44.

Ramesh, B. and Dhar, V. 1992. Supporting systems development by capturing deliberations during requirements engineering. *IEEE Trans. Softw. Engin. 18*, 498–510.

Ramesh, B. and Jarke, M. 2001. Towards reference models for requirements traceability. *IEEE Trans. Softw. Engin. 27*, 1, 58–93.

Robillard, P. 1999. The role of knowledge in software development. *Comm. ACM 42*, 1, 87–92.

Rowen, R. 1990. Software project management under incomplete and ambiguous specifications. *IEEE Trans. Engin. Manag. 37*, 1, 10–21.

Rus, I. and Lindvall, M. 2002. Knowledge management in software engineering. *IEEE Softw. 19*, 3, 26–38.

Schmidt, F. and Hunter, J. 1989. Interrater reliability coefficients cannot be computed when only one stimulus is rated. *J. Appl. Psychol. 74*, 368–370.

Schwab, D.P. 1980. Construct validity in organizational behavior. *Research in Organizational Behavior*. 2, 3–43.

Seaman, C. and Basili, V. 1998. Communication and organization: An empirical study of discussion in inspection meetings. *IEEE Trans. Softw. Engin. 24*, 7, 559–572.

Seeley, D. 2003. Coding smart: People vs. Tools. *ACM Queue 1*, 6.

Shore, T.H., Shore, L.M., and Thoronton, G.C., III. 1992. Construct validity of self- and peer evaluations of performance dimensions in an assessment center. *J. Appl. Psychol. 77*, 1, 42–54.

Spinellis, D. 2005. The tools at hand. *IEEE Softw. 26*, 1, 10–12.

Straub, D.W. 1989. Validating instruments in MIS research. *MIS Quart. 13*, 2, 147–166.

Strauss, A. and Corbin, J. 1998. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, Inc., Newbury, CA.

Takeishi, A. 2002. Knowledge partitioning in the interfirm division of labor: The case of automotive product development. *Organiz. Sci. 13*, 3, 321–338.

Tiwana, A. and Keil, M. 2004. The one minute risk assessment tool. *Comm. ACM 47*, 11, 73–77.

Tiwana, A. and Mclean, E. 2003. Managing the unexpected: The tightrope to e-business project success. *Comm. ACM 46*, 12, 345–350.

Tiwana, A. and Mclean, E. R. 2005. Expertise integration and creativity in information systems development. *J. Manag. Info. Sys. 22*, 1, 13–43.

Todd, J.D. 1979. Mixing qualitative and quantitative methods: Triangulation in action. *Administrative Sciences Quart. 24*, 4, 602–611.

Walz, D., Elam, J., and Curtis, B. 1993. Inside a software design team: Knowledge, sharing, and integration. *Comm. ACM 36*, 10, 63–77.

Weekley, J.A. and Gier, J.A. 1989. Ceilings in the reliability and validity of performance ratings: The case of expert raters. *Acad. Manag. J. 32*, 1, 213–222.

Yin, R.K. 1994. *Case Study Research: Design and Methods*. Sage Publications, Newbury, CA.

Received August 2005; revised March 2006; November 2006; February 2007; accepted March 2007