



The Determinants of Individual Performance and Collective Value in Private- Collective Software Innovation

**Ned Gulley
Karim R. Lakhani**

Working Paper

10-065

Copyright © 2010 by Ned Gulley and Karim R. Lakhani

Working papers are in draft form. This working paper is distributed for purposes of comment and discussion only. It may not be reproduced without permission of the copyright holder. Copies of working papers are available from the author.

The Determinants of Individual Performance and Collective Value in Private-Collective Software Innovation

by Ned Gulley (The MathWorks) & Karim R. Lakhani* (Harvard Business School)
Version 2.0: February 7, 2010

Abstract:

We investigate if the actions by individuals in creating effective new innovations are aligned with the reuse of those innovations by others in a private-collective software development context. This relationship is studied in the setting of eleven “wiki-like” programming contests, where contest submissions are open for reuse by others, each involving more than one hundred contributors and several thousand attempts to generate, over a one-week period, the “best” software solution to a difficult programming challenge. We find that greater amounts of new code and novel recombinations of others’ code, in a contest submission, increases both the probability of achieving top rank and the subsequent reuse by others in their own submission (community value). While, increasing use of borrowed code in a submission reduces the probability of achieving top rank, but increases the community value of the submission. Code structures that are more non-conforming to commonly accepted programming conventions similarly increase the probability of generating a top performer, but reduce subsequent reuse by others. Surprisingly, greater code complexity in a submission increases both the odds of generating a top performing entry and its community value. We discuss the implications of these findings in light of the literature on private-collective innovation with an emphasis on the importance of considering both individual and community perspectives as they relate to knowledge creation, reuse and recombination for innovation.

***Please send comments to klakhani@hbs.edu**

1 Introduction

Over the past decade, collaborative, community-based approaches to developing knowledge-intensive products like encyclopediae, music, and software have gained prominence in both practice and scholarly analysis (Benkler 2004; von Hippel 2001; Weber 2004). In particular, we have seen numerous collaborative communities focused on developing complex software emerge across and outside of formal organizations (Baldwin and Clark 2006b; Feller et al. 2005; Lakhani and von Hippel 2003; Roberts et al. 2006; Stewart and Gosain 2006). This “open source software development” is distinguished by self-selection of distributed participants into tasks, free revealing of knowledge, collective creation of shared software artifacts, and participants’ ability to generate new innovations by reinterpreting and repurposing knowledge and artifacts created by others. A distinguishing feature of this “private-collective” innovation model (von Hippel and von Krogh 2003) is the exertion of private effort, without external subsidies, towards the co-creation of a public good.

Core to the theoretical viability and empirical persistence of the private-collective innovation model are two related propositions about the drivers of public goods creation. First, the free-rider problem, noted in most commons settings, is overcome by the attainment of private selective benefits in the process of contributing to the public good. Hence private effort results in private selective benefits not attainable by free riders, thus motivating public contributions (von Hippel and von Krogh 2003). Second, that the public goods being created will in fact be reused, recombined and made better by the efforts of others and that collective value is enhanced when contributions are reused more often by others (Murray and O’Mahony 2007). In this paper we examine if there is alignment of individual actions in both accruing private benefit and the generation of collective value in a private-collective innovation system.

Specifically we investigate two research questions. How do individual choices in the sources of knowledge used to generate software code, and the design structure of that code, affect participants’ technical performance and related private benefits?¹ How do those same choices, made by the individual, enhance collective value for others, that is, how often is new knowledge contributed by an individual reused by other participants? We answer these questions empirically by using data from a community-based software development setting, a particularly relevant context because code as text represents the prescriptive knowledge (Mokyr 2002) of its creators and code as a technical artifact takes on a particular design structure and has

¹ Individuals creating software code in a private-collective setting can invent new knowledge (code) and reuse and/or recombine in novel ways their own or others’ existing knowledge. Such knowledge is then organized in a particular technical form, namely, the code itself, that mediates its performance.

performance consequences (Baldwin and Clark 2006a; Haefliger et al. 2008) which adjudicates the private benefits.

Addressing these questions is relevant because both the information systems and innovation literatures have separately identified the importance of effective knowledge creation and reuse (Argote et al 2000, Markus 2001), the design and architecture of technological artifacts (Banker and Slaughter 1997; Banker et al 1998, Baldwin and Clark 2000), and the emergence of collaborative and open approaches to technology development (von Hippel 2005). Indeed, because the *development* of information systems and software can be conceived more generally as an innovation process involving human problem solving and creativity, and as more sectors of the economy evaluate private-collective innovation models (e.g., academics and practitioners in biology and pharmaceutical sciences are pursuing these models, Hess and Ostrom 2003), understanding the emergence of these new approaches is becoming more germane to scholars and practitioners alike.

We study the determinants of individual performance and collective value in a private-collective software innovation setting by analyzing a unique dataset of eleven, The MathWorks corporation-sponsored programming competitions. These week-long competitions challenge developers to code in the MATLAB programming language solutions to NP-complete² mathematical problems (such as the “traveling salesman” puzzle) elicited, per competition, between 1,631 and 6,933 entries from 94 to 199 programmers from around the world. Code submitted throughout a seven-day period is immediately evaluated against a test-suite that rates algorithmic accuracy (in the case of the foregoing problem, minimizing travel distance) and computational efficiency (minimizing CPU time utilization) and generates a composite score and rank for each submission. The code that earns the lowest score at the time of submission is ranked first, and so on. At the conclusion of the competition, the participant with the lowest overall score for submissions wins a nominal prize (typically a t-shirt). A unique feature of these contests is that, after the first two days of the contest, all code submissions are open for examination and reuse by anyone participating in the contest. Thus a purely competitive contest is transformed into a community effort. We focus our study on the five day open period in the contests.

Our analysis reveals both alignment and tension between individuals’ need to create top ranking submissions and their subsequent reuse by others. Achieving top rank is predicted by

² A problem is NP-complete if its answer can be determined quickly and an algorithm to solve this problem can be used to solve all other NP problems quickly.

greater numbers of new lines of code, fewer numbers of borrowed lines of code, and more numerous novel recombinations of borrowed code. Submissions that have more complex code and disregard commonly understood coding conventions are more likely to be among the top ranked. The collective value of a submission, that is, how often lines of new code are reused by others, is related to increasing numbers of both new lines of code generated, and lines of others' code borrowed, by a participant. More numerous instances of novel recombinations of existing code also enhance community value. Surprisingly, increasing complexity did not have the anticipated effect of reducing the collective value of code, instead, was found to be a significant predictor of its further reuse. Non-adherence to coding conventions however, diminishes the collective value of code. Hence, we report a tension between the needs of individuals and those of the many with respect to the use of borrowed code and conformance to accepted coding practices.

Our findings contribute to the literature by first highlighting the contradictory affects of free-riding in private-collective innovation systems. In our context, free riding, or borrowing code, has both negative and positive implications for individual performance and collective value. For individual innovators, increasing amounts of non-creative borrowing of others' code results in poorer performance, but novel recombinations of borrowed code enhances performance. At the same time, the collective value of new code contributions increases in direct relationship to the amount of both non-creative borrowing and novel re-combinatorial borrowing. Thus while non-creative borrowing is typically not encouraged in private-collective innovation theory – it does play an important role in helping to increase the reuse of new contributions as community members' assessment of the value of new contributions is directly impacted by the presence of previously submitted contributions. Equally importantly, we provide evidence of the positive role of re-combinatorial free-riding, creating novel recombinations of existing knowledge of others, for both individual performance and collective value.

The contradictory impact of non-creative free riding is further bolstered by our findings on the role of increasing code conformance, i.e. implementing code in commonly acceptable protocols, in predicting poorer individual performance and superior collective value. Our findings show that collective value relies on the ability of others' to understand and comprehend the design structure of knowledge to enable reuse. Thus deviations from commonly understood rules of practice, while beneficial to the individual innovator, impede adoption by others. Hence we draw attention to another tension between individuals and other community members, not previously raised in private-collective theorizing, and raise the issue that adoption and reuse of

the public good maybe end up at cross purposes with the pursuit of private benefits.

Our findings about the factors predicting collective value are also consistent in direction and significance if we predict just the first adoption by someone else in the community. However, when we focus our attention on predicting collective value in just the top performing submissions we discover that factors like code borrowing, novel re-combinatorial borrowing and complexity, while consistent in sign, are not significant. Thus raising the question for designers and scholars of private-collective innovation systems if general reuse of knowledge is preferred, regardless of performance, or attention solely needs to be paid to simply pushing the technical frontier.

The rest of the paper is organized as follows. We review, in Section 2, the literature on private-collective innovation and, in Section 3, details of the empirical context in which we study specifically private-collective software innovation. The hypotheses that guide our analysis on the basis of the knowledge sources and artifact structure used by a contributor in a private-collective setting are generated in Section 4. We report our data sources, variable construction, and estimation approach in Section 5, and findings and limitations in Section 6. We discuss our findings and present our conclusions in Section 7.

2 Private-Collective Software Innovation

The dominant academic and practitioner view of technology development, in general, and software development, in particular, assumes a private investment model whereby society grants individuals and firms limited monopoly rights in return for investments in and efforts at generating innovations (Demsetz 1964). Investments in innovation are made in the interest of achieving private gain in the face of some risk of knowledge spillover consequent to participating in the marketplace. At the same time, society favors incentives for private creation over the social loss incurred in creating knowledge monopolies (von Hippel and von Krogh 2003).

An alternative collective-action model, historically dominant in the sciences, envisages innovation effort outcomes as non-rival, non-excludable public goods whereby knowledge is contributed to a common pool to be reused by others (Dasgupta and David 1994; Merton 1973). At the heart of the collective-action innovation model is the dilemma of free riding. Because in a collective action system all outputs are public goods, it would be rational for participants to neither invest nor publicly disclose their contributions and wait for others to contribute. Society fosters innovation in collective systems by providing financial grants, subsidies, and other

incentives to encourage individuals and organizations to embrace public disclosure of the effort they exert and unconditional reuse of the fruits of their labor. Participants in such systems self-regulate sharing behavior by creating norms of reciprocity and recognition whereby the success of originators of high quality contributions is acknowledged through peer recognition and ongoing reuse of their work (Dasgupta and David 1994; Stephan and Levin 1992).

2.1 The Emergence of Private-Collective Models of Innovation

Whereas these innovation systems have operated primarily in parallel, historical studies of technology development during the industrial revolution have convincingly demonstrated the existence of a hybrid private-collective model of innovation (Nuvolari 2004; Osterloh and Rota 2007). Allen (1983), for example, showed that the mid-nineteenth century innovation process underlying the critical blast furnace technology for iron-making involved simultaneous private investments and public disclosure of the knowledge behind and improvements to the inventions. This “collective invention” process gave rise to dramatic improvements in furnace height (from 50 to 80 feet) and operating temperature (from 600 to 1,400 Fahrenheit), which significantly reduced fuel consumption in, and increased the profitability of, iron production. Similar examples of collective invention have been observed in the development of Cornish pumping engines, Bessemer steel, and large-scale silk production (Nuvolari 2004).

Software development has itself gone through phases of collective and private innovation models. Before the advent of commercial software firms, computer hardware users tended to be university researchers and commercial research and development units that viewed software as a research tool to be developed and improved by all users (Campbell-Kelly 2003). In its early days as a hardware company, IBM distributed its software code with minimal restrictions and encouraged collective modification and improvement by users; indeed, its software user group was called SHARE³ (Campbell-Kelly 2003). The private investment model of software development has dominated since the early 1960s, when a number of entrepreneurs began to realize that software could be sold independent of hardware and to recognize the potential for profiting, as computer use expanded throughout the economy, from developing and creating standardized software that could serve the needs of many less sophisticated users.

The enormous commercial profits generated in the software industry notwithstanding, the notions of sharing code and jointly creating a public good was never really suppressed. Unix in its various flavors initially emerged in 1970s, under the aegis of the Bell research laboratories, as

³ SHARE was not an acronym.

a fully user-developed operating system that was freely shared and modified (Salus 1994). The Free Software Foundation (FSF), founded in the 1980s to encourage, in response to its increasing “privatization,” the creation of software as a public good (Stallman 2001) hired programmers and solicited volunteers to contribute time and effort to developing freely available software. The FSF pioneered intellectual property regimes that ensured that software that became a public good could not be easily privatized. The waxing of the Internet in the 1990s enabled users and developers the world over to find one another and, using simple tools, collaborate on the creation of software as a public good, an endeavor that became known as the open source software movement (Raymond 1999). The engagement of hundreds of thousands of individual and firm-sponsored volunteer programmers in multitudes of software projects has given rise to an enduring economic and sociological puzzle posed by Lerner and Tirole (2002) as: “Why should thousands of top-notch programmers contribute freely to the provision of a public good?”

2.2 Unpacking the Private-Collective Innovation Model

Von Hippel and von Krogh (2003) first articulated the hybrid private-collective innovation model as a way for scholars to understand the open source software phenomenon. The creation of innovations subsequently reused by others is critical to the model, which assumes the existence of an intellectual property (IP) regime in which the knowledge and artifacts created by innovators are publicly disclosed, that is, made known and rendered fully accessible (i.e., available for reuse and recombination) to others (Murray and O'Mahony 2007; Stuermer et al. 2009). This IP regime does not, however, guarantee that private contributions to the public good will be made, that is, that the free rider problem will be overcome.

2.2.1 Selective Benefits of Contributing to Public Goods

Von Hippel and von Krogh (2003, p. 216) posit that the puzzle of private creation of public goods can be solved by recognizing that “contributions to open source software development are not pure public goods—they have significant private elements even after the contribution has been freely revealed.” The authors first question the private investment model’s assumption that free revealing of innovations developed with private investment and effort constitutes a loss for the innovator(s). They find instead that under some conditions free revealing earns innovators a net benefit in terms of others’ suggested improvements to, as well as adoption and diffusion of, the original contribution. Von Hippel and von Krogh (2003) also challenge the assumption of the collective action model that benefits derived from freely revealed contributions are spread homogeneously, among free riders as well as innovators, finding instead that innovators enjoy

access that free riders may not to highly prized selective benefits. The core of the private-collective innovation model is thus the accrual of such selective private benefits to the individuals who expend, at relatively low cost, effort to create non-rival, non-excludable public goods.

Both intrinsic and extrinsic motivations have been proposed as drivers of individual participation in the private-collective model of innovation (Hertel et al. 2003; Lakhani and von Hippel 2003; Lakhani and Wolf 2005; Roberts et al. 2006; Stewart and Gosain 2006). Intrinsic motivations include psychological benefits such as enjoyment and the stimulation associated with intellectual challenge and satisfaction derived from completing a task at hand (Amabile 1996; Csikszentmihalyi 1990; Ryan and Deci 2000). Individuals involved in studies of open source software development self-report as reasons for their contributions relatively high levels of intrinsic motivation (Hertel et al. 2003; Lakhani and Wolf 2005; Roberts et al. 2006), but empirical findings on the correlation between intrinsic motivations and participation levels are mixed. Thus, neither Hertel et al.'s (2003) study of hours spent by Linux kernel developers on a project, nor Roberts et al.'s (2006) study of contributors' source code contributions to the Apache project, surfaced a significant relationship between intrinsic motivators and participation, whereas Lakhani and Wolf's (2005) study of lower profile open source projects on SourceForge.net found the sense of personal creativity individuals derived from involvement in project activities to be a significant predictor of time spent contributing.

Extrinsic motivations can be direct or indirect. Third-party payment for participation and user need for a product are obvious instances of direct motivators (Lakhani and Wolf 2005; Roberts et al. 2006; von Hippel 2001), job market signaling, accruing community and professional status, and learning by doing and through peer review examples of indirect motivators (Lakhani and Wolf 2005; Lerner and Tirole 2002; Roberts et al. 2006).

Empirical studies of open source software have shown extrinsic motivation to play a significant role in predicting levels of individual participation in a project. Not surprisingly, Lakhani and Wolf (2005) and Roberts et al. (2006) found, respectively, being paid to be a significant predictor of hours per week spent on a project and number of source code contributions, and Hertel et al. (2003), Lakhani and Wolf (2005), and Roberts et al. (2006) find significant support for a positive role for status concerns in participation levels. Community status is positively related to number of hours spent on a project (Hertel et al. 2003; Lakhani and Wolf 2005) and professional and community status to number of code submissions (Roberts et al. 2006). Among intrinsic and extrinsic motivations for participating in private-collective

innovation, status seeking in community and professional circles tends to predict higher levels of participation.

2.2.2 Reusing Others' Knowledge in Private-Collective Innovation

As important as motivating contributions to private-collective innovation is that they be reused, recombined, and improved upon by other members in the collective (Murray and O'Mahony 2007). Reuse as well as creation of knowledge are thus core components of the private-collective innovation model. Nor is the concern for knowledge reuse limited to collective or private-collective contexts. A central area of interest to scholars and practitioners alike has been the practice and extent of knowledge reuse in private settings like firms (Argote et al. 2003; Majchrzak et al. 2004) to introduce efficiencies into and improve the quality of internal innovation processes (Banker et al. 1991; Knight and Dunn 1998).

The dual nature of software, as both explicit knowledge in the form of text (source code) and electronic machine instructions, has spurred extensive studies and inquiries into the nature of knowledge reuse in firm-specific software development settings (Haefliger et al. 2008). Such important benefits as cost savings, reduced development time, and higher quality notwithstanding, software reuse within firms has found to be at sub-optimal levels and, where employed, to be problematic (Fichman and Kemerer 2001; Fichman and Kemerer 1997; Pérez and Benjamins 1999; Rothenberger et al. 2003). Ironically, managers for whom free riding is not a concern and who have direct authority over employees nevertheless struggle continuously to encourage developers to reuse code developed by their colleagues.

Software reuse in private settings is driven by a simple logic of cost minimization, that is, whether search and integration costs associated with reusing code developed by others are lower than the cost of denovo coding (Banker et al. 1994; Banker et al. 1998; Haefliger et al. 2008).⁴ The literature has identified as essential to software reuse the following four elements: infrastructure for code repositories; quality assessment; technical structure; and individual incentives.

Absent an adequate infrastructure for storing, classifying, searching, and retrieving code, developers will waste time and effort searching for components (Poulin 1995) and thus reuse will be minimized. Developers who find appropriate code need assurance that it is defect free and will not adversely affect the programs in which it is to be used (Lynex and Layzell 1997, 1998). A “believable” assessment of the quality of available reusable software components is thus

⁴ See Levine and Prietula (2006) for a general model of the costs and benefits of knowledge transfer.

needed. Developers must also understand, having found a suitable, high quality software component, how it works and what modifications are needed to it or to the program into which it is to be integrated. Technical structure is relevant here, as code that is overly complex or does not follow commonly accepted software writing conventions is less likely to be reused (Rothenberger et al. 2003). Lastly, the stigma that attends activities related to code reuse, as being somehow inconsistent with desirable traits of a good programmer, need to be overcome. Many developers are disinclined to reuse code because they construe doing so to be “copying,” and that to be a boring, non-creative activity that deprives them of a personal sense of problem solving and achievement (Lynex and Layzell 1997, 1998; Poulin 1995).

What of code reuse in a private-collective setting absent managerial and authoritarian edicts? Haefliger et al. (2008), who have conducted a systematic study of code reuse in an open source software setting, found the proclivity to reuse code from the commons to be driven by a need to rapidly develop additional functionality in burgeoning projects and by opportunities to avoid writing “boring” code and to conserve project resources by avoiding reinventing the wheel. Candidate code was assessed in terms of quality, proxied by its popularity in the open source community, and compliance with standards and modular interfaces. Sojer and Henkel (2009) in their study of 686 SourceForge.net developers find that individual developer characteristics like large personal networks and extensive experience in open source projects drive reuse proclivity. Interestingly they also find that those developers that are motivated to solve difficult technical challenges are most likely not to reuse code from other developers.

3 MATLAB Programming Contests

We study how individual actions to obtain private benefits are aligned with collective value generation by analyzing a unique data set of eleven programming contests sponsored by The MathWorks corporation (Gulley 2004). Held approximately every six months, these week-long, web-based contests challenge participants to develop software code in the MATLAB language to solve an NP-complete (Kendall et al. 2008) mathematical problem (e.g., the “traveling salesman” problem) in any of a range of domains including biology, supply chain management, mathematics, and physics. Participants are provided a detailed problem statement, a limited test-suite that enables them to privately evaluate their code writing efforts, and access to a Web interface by which to submit code to the contest scoring engine. The automated scoring engine, different from and more expansive than the limited test-suite available to the participants, evaluates each entry in terms of algorithmic accuracy (in the case of the traveling salesman

problem, for example, travel distance) and computational efficiency (e.g., CPU execution time).

Participants can submit code to be scored as often as desired. The contest Web site maintains a dynamically updated leader board with scores and rankings of each code submission and associated author. The submission that earns the lowest score (in the case of the traveling salesman problem, minimum travel distance and execution time) at the end of the seven-day competition wins a nominal prize (typically a t-shirt). Frequent submissions are encouraged by awarding lesser prizes for various performance metric-related objectives (e.g., most improvement in a day, best entry on a given day, greatest distance minimized in a day). Participants, who tend to be experienced programmers who use MATLAB in industrial, research, or academic settings, covet these interim awards, in particular, attaining the top rank (i.e., being number one on the leader board) for however brief a time. According to contest administrators, participants, just as in pure open source projects, explain their involvement in terms of a combination of intrinsic and extrinsic motivations. Having fun, learning and the intellectual challenge of solving an interesting problem is often mentioned in the contest forums, and equally important is the distinction of achieving the number one rank upon submission of code, throughout the contest, as this signals unique ability and skills amongst elite coders. Being named the contest winner is, of course, important as well.

A unique feature of these contests is the rule that dictates automatic information disclosure. A typical contest has three phases: dark, twilight, and light. Participants can join (or leave) the contest at any time, in any phase. During the one-day “dark” phase, scores of submitted code are revealed only to the submitting participant. During an ensuing one-day “twilight” phase, scores and relative ranks generated by submissions are publicly viewable. During the five-day “light” phase, which is “wiki-like” in the sense that all submissions (including those made during the dark and twilight phases) are made public and transparent upon submission, all participants are afforded access to all submitted code (the scores and ranks of which are at this time known), which can be tapped for insights or incorporated directly into code being written. The amount of such “borrowing” is not constrained by any rule. The inclination to hoard or delay to the last minute submitting code are countered by the condition that performance is fully evaluated only when an entry is submitted to the scoring engine, at which point the code becomes public. The light phase effectively transforms a competitive and proprietary contest into a private-collective innovation system in which both individual and collective performance are prized. Our study focuses in on the five-day light period where all code is freely available.

3.1 Contest Vignette

The following provides a vignette of a contest that ran from May 9th to May 16th, 2007 in which the challenge was to code software to play a modified Peg Solitaire game (one commercial version of which is known as Hi-Q). Traditional Solitaire, in which one peg is jumped over another to an adjacent hole and the jumped peg removed until all pegs are removed save one that should be in the center of the board, has been shown to be an NP-complete problem (Uehera and Iawata 1990). The game as modified for the contest assigned weights and values to “virtual” pegs on an $n \times n$ board and imposed a general rule that lower valued pegs should be used to jump over higher valued pegs, with scoring such that each move accumulated points that included a removal bonus (removed peg value - jumping peg value) and lifting penalty (jumping peg value). A final score was obtained for each board by subtracting the points accumulated from the moves from the sum of the initial value of all pegs on the board. The objective was to achieve the lowest average score across all the game boards in the test suite in a minimum amount of time. Entries that exceeded three minutes were disqualified.

As can be seen in Figure 1, 110 participants submitted 3,914 code entries during the week-long contest. Four hundred eighty eight (a little over 12 percent) of the code entries failed due either to time-out or compilation errors. Figure 2 plots the distribution of the scores of passing entries. Note that the lower the score, the better the performance, entries on the red (dark) line being the top ranked submission at any given time. The top rank was achieved by 180 entries from 26 participants. Analysis of the winning entry, submitted by Yi Cao, provides a snapshot of the contest dynamics. A senior lecturer in the department of Process and Systems Engineering at Cranfield University in the United Kingdom, Cao submitted to this contest, 69 entries of which only five, including the winning entry, achieved the top rank. The striking element of the latter entry, was that it consisted of 545 lines of MATLAB code, of which only 114 lines were originally authored by him and only 12 lines were newly written. The remaining 431 lines, which accounted for 79% of the final submission, were authored by 30 other participants. In a post-contest analysis message to the contest community, Cao acknowledged the following important contributions from others that he incorporated in his entries:

Nick Howe discovered an entry, Juno37, submitted by Jim Mikola in the twilight. The newly discovered solver, solveri, was able to provide better results for certain cases, although it took a longer time. Nick Howe successfully combined solveri with the dominant solver, subsol, in his submission, JuChimera, to win the Saturday Early Morning Special.

Nathan Q submitted his entry, bigh_L8, at 06:12 for the Big Sunday Push. He developed a

way to update the list of all valid moves without re-calculation of the whole list (moveid1~3). This improvement overcame the computation bottleneck of subsol and significantly reduced CPU time by 9 seconds.

Markus Buehren introduced another improvement to simplify the calculation of CV.

I combined several improvements to Nathan's and Markus' codes into a single entry, collaboration solver. Unfortunately, this entry did not go to the top. It was beaten by other randomly tweaked entries. Luckily enough, within half an hour it was discovered by Jin. He made it become the top entry, collaboration solver b. Later, it was taken over by Sergey Yurgenson and helped him to win the Tuesday Leap.

[Near the end of the contest]: I found David Jones' entry, which introduced a magic number to adjust the random sequence, rand(57); then I found MarkR's entry, which included a new set of dlist. Both were leading entries. Hence, I copied both into all my submission pages. Before that, I also noticed Jin submitted Iamcrazy prob series, which detected a set of specific conditions to call solveri. I used these conditions in some of my submissions as well. Finally, about five minutes before the end, I clicked all submit buttons, but the last one was rejected by the sever with a message the queue was closed.

He further noted that:

Peg Solitaire is an interesting contest. The problem is simple and easy to be understood. Anyone with reasonable MATLAB knowledge should be able to develop a valid solver. However, the problem is also so hard that after seven days of hard work many contestants believe we are still far away from the optimal solution.

Reflecting on his general strategy, he added:

This seems to be my style of participation: a lazy brain does not know how to solve the problem at first, then starts looking around trying to understand what others do and to identify room for improvement. Finally the brain cannot stop itself from analyzing the problem even after the contest ends.

The MathWorks contest data provides a unique, social “Petri dish” setting typically not available in conventional field settings in which to closely study the emergence and operation of private-collective innovation systems. That contest participants are focused on solving the same innovation problem enables us to compare many individuals without worrying about task comparability. Although there may be in open source projects multiple parallel attempts to solve the same problem, most projects try to extract the most value from participants’ time by encouraging the generation of complementary modules (Haefliger et al. 2008; von Krogh et al. 2003). As well, the presence of a common test suite creates an objective criteria for technical performance against which all code submissions can be compared. Focusing on the technical

performance of many code submissions for the same problem enables us to overcome concerns about how to measure “success” in an open source context (Crowston and Scozzi 2002). The setting is also relatively controlled in that we can observe the entry and exit of, and effort expended by, each participant as well as the ranking of submitted code, thus minimizing concerns over authorship and commit authority that may be found in open source settings (Roberts et al. 2006; von Krogh et al. 2003). We also are afforded fine-grained tracking of individual lines of code that enables us to precisely observe both invention and reuse. Finally, in its appeal to individual motivations to succeed and achieve private benefits (e.g., high rank) and concurrent emphasis on encouraging knowledge reuse to generate collective value, the contest design provides near laboratory conditions for participation while retaining the virtues of field research.

Section 4: Consistency of Actions in Pursuing Individual Extrinsic Benefits and Generating Collective Value

Consistent with earlier theorizing, we argue that in a private-collective software innovation settings individual programmers will attempt to maximize private benefits in the process of creating a public good. Although intrinsic motivations are important in predisposing individuals to participate, the level and intensity of participation are driven primarily by a desire for status and reputation in the community of peers. As differentiation, reputation, and status typically accrue to the demonstrable creation of superior technical solutions to the problem at hand, pursuit of top-level technical performance will be a major driver of individual participation in private-collective software innovation, and actions taken in the software development process in pursuit of top-level performance will have a direct bearing on the generation of collective value. We thus focus our attention on the individual pursuit of top performance and resulting impact on collective value.

Our analysis is grounded in the fact of software development being a design process (Baldwin and Clark 2006b) whereby individuals generate or reuse prescriptive knowledge (Mokyr 2002) in a specific structure (or architecture) to solve a particular problem ((Baldwin and Clark 2006a; Simon 1962). Prescriptive knowledge (i.e., knowledge about how to do something), as instantiated in designs, lies between general propositional knowledge about the world (i.e., the sum of everything known and stored, physically, cognitively, and metaphorically) (Mokyr 2002) and the completion of tasks in the economy (Baldwin and Clark 2006a). In the case of software, inspection of code (i.e., the artifact created in the design process) can reveal both the underlying prescriptive knowledge and specific structure that enables a computer to

perform certain functions that solve a particular problem. Measuring the technical performance of software code against specified objectives provides an assessment of the quality of the knowledge and design bundle generated by a programmer, thereby enabling relative comparisons against other attempts and rendering software designs rankable (Baldwin and Clark 2006a, pg 11). The extent to which newly created prescriptive knowledge is adopted by others provides a signal about the generality and scope of usefulness of a solution. Thus, our analysis focuses on two levers available to individual programmers, (1) sources of prescriptive knowledge, and (2) the design structure of the software artifact produced.

4.1 Sources of Prescriptive Knowledge

Innovation, as classically defined, implies the creation of new knowledge and its implementation and use in the marketplace (Afuah 1998). Indeed, the primacy of new knowledge in innovation is enshrined in the United States constitution and legal code whereby patents, state-granted knowledge exercise monopolies, are to be granted only to inventions that are “novel” and “non-obvious.” New ideas and the realization of things that did not previously exist are elemental outcomes of innovation (Downs Jr and Mohr 1976).

In the context of software development, new prescriptive knowledge takes the form of new code, contributions of which are the sine qua non of a private-collective innovation system in a public goods setting, and represent attempts by programmers to creatively solve problems and simultaneously achieve top performance among peers.

Hypothesis 1A: Greater amounts of new code in an entry increases the probability of achieving top rank upon submission.

It follows that the more new prescriptive knowledge in a submission, the more likely it will be used by others and enhance collective value.

Hypothesis 1B: Greater amounts of new code in an entry, controlling for technical performance, increase its collective value, that is, the likelihood new code will be reused by others.

The desire to generate new code in a private-collective innovation setting is tempered by the availability of existing code that can be tapped for well-understood ideas generated by others or used to economize on effort or avoid tackling “boring” problems or elements thereof (Haefliger et al. 2008). Too much reliance on existing knowledge, however, can engender

stagnation and obsolescence (Sorensen and Stuart 2000) and limit achievement, at best, to performance levels previously attained by others.

Hypothesis 2A: Greater amounts of borrowed code incorporated in an entry will diminish the probability of achieving top rank upon submission.

From a collective value point of view, increasing use of existing code diminishes the relative value of new contributions of prescriptive knowledge, which may not be valued as highly by potential adopters when embedded in a strata of code created before by others.

Hypothesis 2B: Greater use of borrowed code in an entry, reduces the collective value of the new code.

New knowledge does not exist in a vacuum, however, and is often defined in relation to existing knowledge. Empirical studies of innovation have shown novelty to exist on the substrate of existing and old knowledge, the so-called “standing on the shoulders of giants” effect (Scotchmer 1991). This view of the symbiotic relationship between old and new knowledge sets is perhaps best articulated in terms of the potential to generate breakthrough innovations through novel recombinations of old and existing knowledge with new perspectives (Fleming 2001; Katila and Ahuja 2002; Schumpeter 1934). This is not a random process of mixing two irrelevant ideas, but rather presumes a significant effort to understand the performance implications of joining together disparate prescriptive knowledge sets (Fleming 2001). Programmers able to achieve this in a private-collective setting are likely to be among the top performers.

Hypothesis 3A: Greater use of novel recombinations of borrowed code, i.e. re-combinatorial borrowing, will increase the probability of an entry achieving top rank upon submission.

To the extent that they attract the attention of other participants, such novel recombinations will drive the adoption of new lines of code embedded in a submission.

Hypothesis 3B: Greater use of novel recombinations of borrowed code, i.e. re-combinatorial borrowing, will likely increase the collective value of new code in an entry.

4.2 Design Structure

All artifacts have an underlying structure that can be represented by the “information shadow” of the various types of interactions among the constituent components (Baldwin and Clark 2006a). In software, in contrast to physical products, there is no separation between the

artifact and its information shadow. The structure of software, according to Baldwin and Clark (2006b, p. 1117), “is not a matter of natural law, but is to a large degree under the control of the initial designer of the system.” Programmers make conscious decisions about the structure of the software artifact during the programming cycle, and their individual choices and preferences can result in significantly different design structures for accomplishing the same objectives with varying levels of performance (MacCormack et al. 2006).

Of the many ways of measuring and assessing the structure of software designs, we emphasize two elementary considerations that have an impact on both technical performance and subsequent reuse, (1) complexity (Simon 1962), and (2) conformance to established software coding practices (Meyers and Lejter 1991). Simon (1962, pg 468) informally defined a complex system as “one made up of a large number of parts that interact in a non-simple way,” and noted that systems designed to accomplish the same task can have very different structures. In the domain of information systems, the complexity of the software artifact has been much studied and debated, generating more than five hundred studies and more than 100 distinct measures of complexity (Zuse and Bollmann 1989). Banker et al. (1998) and Darcy and Kemerer (2002) adopt a theory-based approach to software complexity using Wood’s (1996) task complexity model to identify three types of code-related software complexity constructs, (1) component, (2) coordinative, and (3) dynamic. Banker et al. (1998) operationalize these constructs as follows: component complexity refers to the number of data elements embedded in a software program; coordinative complexity refers to the decisional density in the code-base (i.e., the number of conditional and control statements); and dynamic complexity refers to the volatility of the decisions required as the software executes.

Increasing complexity occasions a tension between the desire for greater functionality, that is, software that is more fully featured and accomplishes more tasks (i.e., exhibits better technical performance), and the ability of others to understand its construction, as increasing the cognitive burden has the potential to retard reuse and modification. Software developers and designers face a tradeoff between optimizing purely technical performance criteria (doing more, faster, better) and economic and organizational concerns for reuse (Hölttä et al. 2005). In a private-collective setting, attempts to maximize private benefits by demonstrating technical superiority in order to achieve high performance will likely occasion more complex design structures.

Hypothesis 4A: The greater the complexity of the code in an entry, the better the technical performance.

Reuse of software code in a private-collective setting is functionally similar to firms' internal efforts to maintain and enhance developed code. Programmers who reuse code developed by others face the challenging cognitive task of identifying and attempting to understand the interdependencies in the code base and discern the thought pattern of the original programmer (Banker et al. 1998). Programmers have been shown to spend as much time studying and trying to understand existing code as writing new code (Fedjelstad and Hamlen 1983), and increased decisional complexity (both coordinative and dynamic) has been shown to increase costs of maintenance (Banker et al. 1998). Thus, the greater the complexity of the source code, the more difficult it will be for others to reuse new knowledge in the submission.

Hypothesis 4B: Greater complexity in the code of an entry will diminish the collective value.

As a human design activity, programming is more art than science, both software and the computers that run it being subject to mistakes occasioned by a mismatch between programmers' expectations and the design of the machine execution system. The mistakes programmers can make as they create their artifacts are of two types, (1) violations of the rules of the language, and (2) failure to follow commonly understood programming conventions. The first is typically caught by the language compiler or interpreter, and results in the software not working. The second can result in code that, although it might execute and narrowly accomplish the task at hand, is fragile, non-robust, and prone to errors.

Owing to conventions that have evolved between programmers and machine execution system developers (e.g., language compilers), in essence, "a distillation of the collective historical experience of the programming community" (Meyers and Lejter 1991), mistakes of the second kind can be automatically identified and tracked. The structure of code design can thus be assessed for potential faults and defects based on a priori agreed upon conformance to established coding standards (Johnson 1978). This type of assessment and flagging is similar to "grammar checking," the commonly used function of modern word processors that assesses and compares to pre-established rules of writing and expression the structure of a corpus of text. A grammar checker can find in a paragraph of text that may be comprehensible to a naïve reader areas for improvement and clarification that will enhance comprehension. There is empirical evidence that the less software conforms to accepted standards, as measured by number of warnings generated by automated conformance tools, the more likely it is to exhibit serious defects (Emanuelsson and Nilsson 2008).

Hypothesis 5A: The less it conforms to code writing standards, the less likely that a submission will achieve top performance.

Similarly, the more prone to mistakes the design structure of submitted code, the less likely that the new prescriptive knowledge embedded in it will be adopted by others.

Hypothesis 5B: The less a submission conforms to code writing standards, the less will be the collective value of new code embedded in it.

4.3 Code Quality and Reuse

Knowledge reuse studies in private and private-collective settings have emphasized that adoption of existing knowledge is contingent on the user's assessment of the quality of the source (Haefliger et al. 2008; Lynex and Layzell 1997; Majchrzak et al. 2004; Pérez and Benjamins 1999). In private-collective settings without mandates to reuse code, adopters will actively search for signals that confirm the quality of code being considered and prefer that which is highly ranked. This leads to our final prediction.

Hypothesis 6: Higher ranked entries will have greater collective value.

Figure 3 summarizes our hypotheses and presents a conceptual model that guides our analysis.

<insert figure 3 about here>

5 Data, Variables, and Estimation Strategy

Our analysis of data from eleven MATLAB contests conducted between 2002 and 2007 focuses on the five-day "light" phase during which participants are free to utilize and reuse others' code submissions as well as create novel code. Our sample includes 26,927 code submissions by 1,306 participants during the light phase of eleven contests. Each contest had, on average, 2,488 code submissions (range: 1,562–4,587).

The web-based structure of the contest provided unusually high fidelity (down to the level of a line of code in an entry) for tracking code submissions and their performance characteristics. We first processed all code submissions through a text parser and emitter to remove comment lines and obvious obfuscations. This yielded a reduced-form database of all lines of code in all entries based on a unique hash table. We were able to track, for each line of code in each submission in each contest, the timing of its first entry into the contest and identify the associated

author and submission that contained it, the performance characteristics of the submission (score, rank at debut), and subsequent reuse of the line of code by other authors and by the same author in other submissions. We were thus able to track for any given submission the number of new-to-the-contest lines of code created by the author, number of lines of code borrowed from the author's previous submissions, and number of lines and source of code borrowed from others. As new lines of code were subsequently introduced, we could track their reuse by others.

5.1 Dependent Variables: Lead Entry and Collective Value

Our analysis includes two dependent variables to account for individual performance and collective value. For individual performance, we examine whether a submission to the contest achieved a debut rank of one and thus became the *LEAD_ENTRY*. This is a binary variable that is equal to one if a submission's debut rank in a particular contest is one, and zero otherwise. Anecdotal interviews and self-reporting by participants confirm this variable to be relevant, with many valuing the top rank and being on the leader board sufficiently to engage in a continuous struggle to achieve the lead rank as the contest unfolds. More than 5.8% of all submissions achieved a debut rank of one. Although the score achieved by a submission is an alternative measure of performance, we prefer to use *LEAD_ENTRY*, as the score calculation is specific to each programming challenge and does not translate across contests.⁵

To gauge the social value of a submission, we create the variable *COLLECTIVE_VALUE*, which tracks the number of times new lines of code in a particular submission are reused by other participants in subsequent submissions. Hence, *COLLECTIVE_VALUE* is a count variable for each submission that increments by one every time a new line of code in the original submission is reused in another participant's submission. As robustness checks, we model *COLLECTIVE_VALUE* in two additional ways later in the analysis. *COLLECTIVE_VALUE_FIRST* counts simply the first adoption of new lines of code by other authors, and does not account for additional reuse of that code; *COLLECTIVE_VALUE_LEAD* counts, in submissions by other authors who achieved a debut rank of one, the reuse of the new lines of code in a submission.

5.2 Explanatory Variables

⁵ We also considered estimating on the basis of ultimate rank received in the contest based on the final scores. This approach does not reflect the dynamics of the contest as experienced by the participants, who value immediate dynamic ranking based on current submissions, i.e., an entry with a rank of ten at the end of the contest is valued less than the entry that achieved a rank at debut of one but ultimately was ranked 500.

The principle variables of interest relate to the sources of knowledge employed in a submission and the structure of the technical artifact, that is, the code itself. *NEW_CODE*, the count of the number of lines of code in a submission that are new to the contest, is our measure of the new knowledge in a submission, *BORROWED_CODE*, the count of the number of lines of code in a submission that have been borrowed from other authors, a measure of the use of existing knowledge in a submission. *NOVELCOMBO_BORROWED*, a pair-wise count of lines of borrowed code that have previously not been together in any other prior entry by any other author, is a measure of the extent to which the pattern of code borrowing by an author is novel. If, for example, four of ten lines of code borrowed by an author from other authors had not previously been used in the same entry, the value of *NOVELCOMBO_BORROWED* would be six.

The code itself is a technical artifact that possesses a specific design structure with implications for both performance and reuse. *CODE_COMPLEXITY* utilizes McCabe's (McCabe 1976) graph theoretic complexity measure of flow control based on the number of linearly independent execution paths in a program. More specifically, the measure considers a program to be a directed graph in which the edges are the decision flow control statements (e.g., IF, CASE, DO, WHILE) and the nodes the straight line segments of code. Subtracting the nodes from the edges and adding two yields the cyclomatic complexity measure (McCabe 1976), which can be considered to be the coordinative decisional/branching complexity of the software code (Banker et al. 1998; Gill and Kemerer 1991). This variable is generated automatically by MATLAB for every function in a software submission. We use the maximum function value reported as our measure of the complexity of a submission.

We also examine the software artifact for conformance to commonly accepted conventions of programming style. The automated static code analyzer "Lint," developed by Johnson (1978), can preemptively identify potential issues in software. Like a grammar checker in word processing applications, Lint can alert programmers to potential errors in the construction of textual artifacts. A program may compile and run and still be poorly constructed, hampering performance and adoption. *CODE_NON-CONFORMANCE* is a count of the number of Lint messages generated during the analysis of an entry by MATLAB. The lower the value of *CODE_NON-CONFORMANCE*, the less likely that there are errors in the submission.

The amount of others' reuse of novel knowledge in a submission may also be driven by the submission's rank. Higher ranked entries may be more likely to be adopted than lower ranked entries. *ENTRY_RANK* records the debut rank of an entry upon submission to the evaluation

engine. The lower the value of *ENTRY_RANK*, the higher the rank.

5.3 Control Variables

We use a range of control variables at the contest, time, author, and entry levels to assist with identification in our analysis. At the contest level, we use dummy variables to isolate differences in the challenges posted in different contests. We also account for the total number of participants in a contest at the time of submission.

Because the contests extend over one-week periods and participation dynamics vary with time zone, time left in the contest, and specific within-contest challenges, we include three controls for time effects. We use the amount of time left in a contest at the time of submission as a control variable. We track the number of submissions in a specific one-hour window to account for any congestion that might be occasioned by mini-contests and end of contest issues, and create a time window counter that increments by one for every hour that passes in a contest.

At the author level, we include controls for the number of previous submissions in a contest and the number of those submissions that achieved a debut rank of one (i.e., occupied the lead position). Finally for each entry we create controls for the number of lines in a submission that were novel from the current leader and the size of the code base. There being a high degree of co-linearity between the lines of code in a submission and number of borrowed lines of code ($r=0.96$), we used the number of distinct operations in the code as a proxy for size. In the collective value analysis, we also account for the number of times lines from the focal entry were reused by the author in future entries until they achieved the lead position in a submission by the author or by another author.

5.4 Estimation Strategy

We use two distinct sets of regression analyses to help us understand the role of knowledge novelty and reuse as it affects individual performance and collective social value. We first use a logistic regression to examine the factors that predict top performance by entries in a contest (i.e., submissions that achieved a debut rank of one). Specifically, we model the probability that a submission is the *LEAD_ENTRY* in Equation 1 as follows:

$$\text{Log} [\text{Pr} (\text{LEAD_ENTRY} = 1 / (1 - \text{Pr}(\text{LEAD_ENTRY} = 1)))] = B_0 + B_1 \text{NEW_CODE} + B_2 \text{NOVELCOMBO_BORROWED} + B_3 \text{BORROWED_CODE} + B_4 \text{CODE_COMPLEXITY} + B_5 \text{CODE_NON-CONFORMANCE} + B_6 i(\text{CONTEST_CONTROLS}_i) +$$

$$B7j(\text{TIME_CONTROLS}_j) + B8k(\text{AUTHOR_CONTROLS}_k) + B9l(\text{ENTRY_CONTROLS}_l) + e$$

(1),

where the subscripts i, j, k, and l refer to various controls related to a particular entry, author, and contest and e is the error term in the estimation. Implementation of the regression clustered the standard errors by participants in a particular contest.

We examine the collective value of a contribution by counting the number of times new lines of code in the current submission were used in other participants' submissions. The standard maximum likelihood Poisson regression model, although used extensively in situations with count-based dependent variables, imposes an equality assumption about the conditional mean and variance that has been shown to be violated in many contexts (Wooldridge 1999), and the negative binomial approach to analyzing count data, although also used in applied research, is liable to produce inconsistent estimates if distributional assumptions about the parameters are mis-specified (Wooldridge 1999). So we follow instead Wooldridge's (1999) guidance and model our count data using the conditional fixed effects quasi-maximum likelihood Poisson estimation, which assumes only that the conditional mean is correctly specified and has been shown to be consistent. Specifically, we model the COLLECTIVE_VALUE count in Equation 2 as:

$$E(\text{COLLECTIVE_VALUE Count}) = \text{EXP} (B0 + B1\text{NEW_CODE} + B2\text{NOVELCOMBO_BORROWED} + B3\text{BORROWED_CODE} + B4\text{CODE_COMPLEXITY} + B5\text{CODE_NON-CONFORMANCE} + B6\text{ENTRY_RANK} + B7i(\text{CONTEST_CONTROLS}_i) + B8j(\text{TIME_CONTROLS}_j) + B9k(\text{AUTHOR_CONTROLS}_k) + B10l(\text{ENTRY_CONTROLS}_l) + e$$

(2),

where the subscripts i, j, k, and l refer to the controls related to a particular entry, author, and contest and e is the error term in the estimation. We used the *xtpqml* Stata module (Simcoe 2007) to derive our estimates for this model. Note that because we are interested in examining how individual actions affect both performance and reuse, the independent variables in both regressions are almost identical, save for the explicit consideration of entry rank as a factor that influences collective value.

6 Descriptive Statistics and Findings

Here we present descriptive statistics that highlight contest characteristics and present the findings of our regression analyses.

6.1 Contest Participation Patterns

Table 1 provides participation pattern details for all eleven contests in our data set. On average, 150 participants (range: 99-199) submitted more than 2,800 code submissions during a typical contest. Approximately 20% of code submissions were disqualified because they either failed to run or exceeded the time limits imposed by the contest rules. Submissions consisted, on average, of 250 lines of MATLAB code of which six were de novo, 211 lines of code borrowed from others and the remaining self-borrowed.⁶ Each submission has 211 pair-wise novel combinations of borrowed code and an average maximum McCabe complexity measure of 30 and generates 20 Lint warning messages for code conformity expectations, and each submitter borrows code from, on average, 16 other participants.

<INSERT TABLE 1 HERE>

6.2 Results

Table 2 contains descriptive statistics for the key dependent and independent variables in the regression analyses. Note that in the regressions, all dependent variables were incremented by one and logged to account for data skewness and facilitate interpretation.

<INSERT Table 2 HERE>

Table 3 provides the coefficient estimates of the logistic regression that predicts that a submission will debut at the top rank (i.e., in the lead position), LEAD_ENTRY. All models include baseline controls (discussed in Section 5.3). The coefficient estimates should be interpreted as the percentage change in the odds of achieving the top rank given a one percent change in the independent variable of interest. Models 3-1 to 3-6 show the impact of each independent variable. In the preceding analyses, each independent variable was entered individually and then with its square term. Only one, CODE_COMPLEXITY, exhibited a significant quadratic effect. Model 3-4 shows the coefficient of CODE_COMPLEXITY to be non significant, but inclusion of the square term, as can be seen Model 3-5, makes the linear and quadratic versions of the variables statistically significant at $p < 0.1\%$. The reduction in the Bayesian Information Criterion value between Model 3-4 and Model 3-5 corroborates the need for the quadratic term in the regression.

⁶ The self-borrowed lines of code are not included in the regression estimation. All lines of code comparisons in the regressions are made in relationship to the self-borrowed lines of code.

Model 3-7, which includes all relevant independent variables, is preferred for interpretation. We find `NEW_CODE` to be positive and significant in increasing the odds of achieving lead status upon submission, thus providing support for Hypothesis 1A. Increasing `BORROWED_CODE`, decreases the odds of achieving a debut rank of one, thus supporting Hypothesis 2A, but increasing novel *combinations* of borrowed code, `NOVELCOMBO_BORROWED`, increases the odds of achieving a debut rank of one, confirming Hypothesis 3A. Hypothesis 4A is supported by the finding that the odds of achieving the lead position upon submission improves with increasing `CODE_COMPLEXITY`. But the quadratic term of this independent variable is negative and significant ($p < 0.1\%$), suggesting diminishing returns to increasing values of `CODE_COMPLEXITY`. Lastly, we find the coefficient of `CODE_NON-CONFORMANCE` to be unexpectedly positive and significant, implying that non-conformance to commonly accepted programming practices (as indicated by the number of Lint messages generated) *improves* the odds of achieving the lead position. Hypothesis 5A is thus not supported, and we find evidence to the contrary.

<INSERT Table 3 HERE>

Table 4 provides the quasi-maximum likelihood fixed effects Poisson regression coefficients on other participants' reuse of new lines of code in a submission (`COLLECTIVE_VALUE`). The coefficient estimates should be interpreted as the percentage change in the count of new lines of code in a submission reused by others, that is, `COLLECTIVE_VALUE`, given a one percent change in the independent variable of interest. All models include baseline controls (discussed in Section 5.3). Models 4-1 to 4-6 shows the impact of each independent variable. We again examined both linear and quadratic terms for each variable. These analyses (not reported here) offered no support for quadratic effects.

Model 4-7, which includes all relevant independent variables, is preferred for interpretation. We find support for Hypothesis 6 in that improvements in `ENTRY_RANK`, a lower number implies higher rank, increase the count of new lines of code reused by others. Although Model 4-2 does not provide support for `COLLECTIVE_VALUE` increasing with increasing use of `NEW_CODE`, including all omitted variables in model 4-7 makes the coefficient of `NEW_CODE` significant at $p < 0.1\%$, thereby supporting Hypothesis 1B.

Surprisingly, the coefficient on `BORROWED_CODE` is positive and significant, indicating that the more borrowed code present in a submission, the more the novel lines of code will be reused by others. This does not provide support for Hypothesis 2B, and we find evidence to the contrary. Models 4-4 and 4-7 confirm hypothesis 3B, that the count of new lines of code reused

by others increases in the presence of greater numbers of novel combinations of borrowed code. Contrary to our expectations, we find the social value of a submission to increase with increasing complexity of the code, `CODE_COMPLEXITY`. Instead of hampering reuse, increasing complexity in fact increases it. We thus cannot confirm Hypothesis 4B, and find evidence to the contrary. Lastly, the negative and significant coefficient of `CODE_NON-CONFORMANCE` confirms Hypothesis 5B, that greater propensity to write code that does not conform to standard practice (i.e., “sloppy” code) diminishes others’ reuse of new lines of code in a submission.

For Model 4-8, which has as its dependent variable the first adoption of new lines of code by others (and does not consider subsequent reuse by the same adopting authors), the coefficient estimates maintain their sign and significance, indicating that the pattern of first adoption is broadly consistent with continual adoption. Finally, in Model 4-9, we consider reuse of novel code among entries that debuted at rank one (i.e., in the lead position). Interestingly, submissions in the LEAD tend to reuse novel code from entries that achieve higher rank and have more novel code and lower Lint message counts, indicating greater conformity. Other independent variables like borrowed code, novel combinations of borrowed code, and code complexity, although of the appropriate magnitude and direction relative to Models 4-7 and 4-8, are non-significant. Figure 4 provides a summary of all of our findings.

6.3 Limitations

We acknowledge a few important limitations to our analysis. Our measure of new prescriptive knowledge is based on new lines of code introduced to the contest. The first concern with this measure is that it fails to acknowledge the general use and simultaneous invention of common programming statements and operations (e.g., “If $i = 1$, do..”) coded by all programmers, crediting them instead to the first person whose submission includes them. We account for this limitation by excluding from our analysis the first two days of the contest, the dark and twilight periods, with the expectation that most of the common programming convention statements will be introduced during these early periods and later inventions will be substantively different. A second concern with this measure is that it counts as new prescriptive knowledge any lexical changes in programming statements (e.g., changing a variable name via a global search and replace operation would increase the count of new code despite there having been no substantive change in programming logic). Although we are unable to overcome this limitation at this time, there are strong norms in the contest community that work against this practice. Individuals who obfuscate code or copy without acknowledgement are singled out and

their code often reintroduced with the variable naming conventions that have emerged in the contest. From an individual perspective, obfuscation would impede the creation of a standard coding convention that would enable a developer to benefit from the work of others.

We are fortunate to have an objective measure of performance in our empirical setting. However this measure is technical and only incorporates algorithmic efficiency and speed resulting in strict vertical differentiation of performance. In other settings, there may be wide horizontal differentiation of performance and in that case “top performing” may not be well defined. Thus we caution extrapolation of our results to other heterogeneous settings. Nevertheless, this vertical performance dimension enables us to provide an in-depth look within a technical domain and assess its consequences and can be generalized to other settings where vertical performance concerns are paramount.

Another potential limitation concerns the population of programmers who participate in these contests. Because they have self-selected on the basis of their interest in and ability to allocate time to these contests, they do not represent a random sample drawn from the population of all MATLAB programmers. Our ability to generalize to a larger population may thus be limited. However, this limitation is inherent in most studies of private-collective innovation systems, as participation, by definition, tends to be based on self-selection, and our sample may be representative of the population of elite MATLAB programmers. Indeed, Levine and Prietula (2008) have argued that studies of private-collective innovation systems should pay more attention to the participants that directly impact the collective outcome instead of the “average” person, who may indeed be free riding or be agnostic.

The average number of lines of code in the submissions in our sample was relatively small, approximately 250 in the MATLAB language. Extending our findings to code bases that are larger and more representative of enterprise environments should thus be done with caution. MATLAB, being a third generation programming language, requires less syntax than first or second generation languages like Fortran or C++. Although objective measures of code equivalence do not exist, users have noted that they can accomplish more with fewer lines of code in MATLAB. Remarked one, quoted in MATLAB marketing materials: “I wrote about 50 pages of FORTRAN code to model a magnetic levitation rail system. When I got all finished, the whole MATLAB program was about eight pages. Now, that’s very typical of what happens when you write programs in MATLAB. The productivity enhancement I feel is about a factor of ten.” CLOC, a program available on SourceForge that counts source lines of code, assigns a 4X conversion factor between lines of code written in MATLAB and lines written in C/C++. Thus,

code submissions in our study are likely representative of code modules that form specialized tasks within larger systems.

Finally, software code, being text that is converted into machine execution, constitutes a unique form of prescriptive and functional knowledge. In most non-coding environments it is not possible to extract the information shadows of artifacts and make them available to large numbers of people for continual improvement. Thus, although many physical products can be simulated in-silico, and similar private-collective innovation systems have been discovered with physical products (e.g., Shah and Franke 2003, von Hippel 2006, Baldwin and von Hippel 2009), we need to be careful about generalizing our findings to non-software environments.

7 Discussion and Conclusions

Private-collective software innovation has turned out to be a fruitful research topic for academics and of significant interest to practitioners in the software industry. A central concern in the research field is the mechanism for motivating private effort in the creation of a public good (von Hippel and von Krogh 2003). In this paper, we utilized a unique field laboratory setting for private-collective software innovation to investigate the consistency of individual actions taken in the pursuit of selective benefits and generation of community value. The MATLAB contest provided an objective measure of technical performance and high fidelity in tracking code generation and reuse to test our hypotheses.

We find both alignment and tension in the actions taken by individuals to achieve top performance and generate collective value. The probability of achieving top technical performance at any given time increases as new prescriptive knowledge and novel combinations of existing knowledge are utilized by participants, and decreases with increasing use of code borrowed from others. Individuals who value the selective benefits of community reputation and career signaling will pursue top performance in a public setting, which is achieved by increasing the generation of new knowledge and novel recombinations of existing knowledge and minimizing free riding as reflected in increased borrowing of knowledge generated by others. Our first contribution is thus to allay the concern that investments in new knowledge may not be generated in a private-collective setting because of the free rider problem.

We also find the collective value of new prescriptive knowledge to be increased when individuals both generate more new knowledge and more novel recombinations of existing knowledge. That collective value is also increased when individuals borrow more from others, that is, free ride, is a source of tension. Reuse of new knowledge occurs only in relation to the

presence of previously generated code. Individual performance is thus improved by less, collective value by more, reliance on free riding. This tension, unanticipated in the private-collective literature, illustrates that in a social system, adoption of new innovations is enhanced when older innovations are incorporated into the new creations. Free riding thus plays a positive role in enhancing the ability of adopters to understand new innovations in light of what has come before. So although free riding is a concern in most collective systems, innovators need to realize that the value of the reuse of their work by others depends as much on the new knowledge the others create as on the old knowledge they borrow. Although standing on the shoulders of giants is not predictive of achieving top performance, it does matter to others who expect to use the public good. Our second contribution is thus to highlight the tension between the need (1) of individuals to achieve top performance, and (2) for others to be able to reuse those individuals' contributions.

We further find that the amount of borrowed code in a submission has a statistically significant positive relationship with reuse of new knowledge in an author's first adoption, but does not affect submissions that subsequently achieve the top rank. This raises the question of whether, in a private-collective setting, only top performing entries (i.e., those that push the technical frontier) should be valued or any reuse is valuable? Conceivably, the ideal is to simultaneously drive widespread reuse and push the technical frontier. In any event, this finding alerts us to be careful of being overly focused on technical performance. General reuse may be useful in the social system, indeed, may be the seeds of top performing submissions.

We also discovered that increasing maximum decisional/coordination complexity in code submissions correlates with both top performance and, unexpectedly, increasing collective value. Greater functionality, proxied by the number of decisional flow control elements in the most complex module, correlates with top performance, as the code is doing more to solve the problem at hand. But this finding is curious in light of abundant theoretical and empirical literature on complexity that finds the cost of adoption to increase as complexity rises (Banker and Slaughter 1997; Banker et al. 1991; Banker et al. 1998). There are three possible explanations for this finding. One, the code base we analyzed was small compared to the systems in which this effect has been observed. However we measure it, complexity may not have the same effect in smaller as in larger systems. This effect is independent of our control for code-base size. Two, our adoption period was a relatively short five days. Adoption generally occurs in longer time cycles, thus, complexity and adoption may have a temporal dimension. Yet another explanation might be that complexity can be trumped by transparency. In our setting, all

actors could view all code submissions and performance information. Complexity concerns might thus be allayed by being able to observe code develop and evolve. Adopters potentially invest more effort in adopting code with greater functionality because doing so benefits their own performance. Pioneering work by Colfer (2009) points to a similar role of transparency in overcoming lack of modularity in open source SourceForge projects.

We also find that writing code that does not conform to established standards yields a personal advantage, but impairs collective value. Conforming to standards incurs both personal and technical costs. It demands attention to conventional norms of programming practice and discourages the taking of shortcuts, even if the resulting code gets the work done, and following proper syntax adds processing time during the execution of code. In a contest in which optimization of algorithms and processing speed are jointly valued, any overhead incurred by following accepted norms of code writing practice may result in a lower score. Yet we find adoption at all levels to be significantly and positively affected by conformance. Thus, even as the incentives to write top performing code mitigate against conformance, adoption is dependent on it. This further corroborates the social-cognitive aspect of code adoption whereby programmers' ability to understand new code is contingent on the style of the code writing.

We contribute to the literature on private-collective innovation systems by elucidating the tensions and alignment between the actions of individuals and needs of the collective. Our study also has implications for the knowledge-based view of organizations and firms. We show knowledge creation and reuse to be important dual goals of social systems organized to solve problems. Theory building and testing in this literature needs to examine concrete knowledge artifacts and trace the process by which they are created and reused. Highlighting the tensions between incentives to create and reuse knowledge may facilitate more effective designs of social systems that rely on many actors for collective problem solving.

References

- Afuah, A. 1998. *Innovation Management: Strategies, Implementation and Profits*. Oxford University Press, New York.
- Allen, R.C. 1983. Collective Invention. *Journal of Economic Behavior and Organization* **4**(1) 1-24.
- Amabile, T.M. 1996. *Creativity In Context*. Westview Press, Boulder, CO.
- Argote, L., B. McEvily, R. Reagans. 2003. Managing Knowledge in Organizations: An Integrative Framework and Review of Emerging Themes. *Management Science* **49**(4) 571-582.
- Baldwin, C., K. Clark. 2000. *Design Rules: The Power of Modularity*, Vol 1. MIT Press, Cambridge, MA.
- Baldwin, C., K. Clark. 2006a. Between 'Knowledge' and 'the Economy': Notes on the Scientific Study of Designs. B. Kahin, D. Foray, eds. *Advancing Knowledge and the Knowledge Economy*,. MIT Press, Cambridge, MA.
- Baldwin, C., K. Clark. 2006b. The Architecture of Participation: Does Code Architecture Mitigate Free Riding in the Open Source Development Model? *Management Science* **52**(7) 1116.
- Banker, R., S. Slaughter. 1997. A Field Study of Scale Economies in Software Maintenance. *Management Science* **43**(12) 1709-1725.
- Banker, R.D., H. Chang, C.F. Kemerer. 1994. Evidence of Economies of Scale in Software Development. *Information and Software Technology* **36**(5) 275-282.
- Banker, R.D., S.M. Datar, C.F. Kemerer. 1991. A Model to Evaluate Variables Impacting the Productivity of Software Maintenance Projects. *Management Science* **37**(1) 1-18.
- Banker, R.D., G. Davis, S. Slaughter. 1998. Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study. *Management Science* **44**(4) 433-450.
- Benkler, Y. 2004. Commons-Based Strategies and the Problems of Patents. *Science* **305**(5687) 1110-1111.
- Campbell-Kelly, M. 2003. *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*. MIT Press, Cambridge, MA.
- Colfer, L. 2009. *Three Essays on the Structure of Technical Collaboration*. Unpublished Doctoral Dissertation. Harvard Business School. Boston, MA
- Crowston, K., B. Scozzi. 2002. Open Source Software Projects as Virtual Organizations: Competency Rallying for Software Development. *IEE Proceedings - Software* **149**(1) 3-17.
- Csikszentmihalyi, M. 1990. *Flow: The Psychology of Optimal Experience*. Harper and Row, New York.
- Darcy, D.P., C.F. Kemerer. 2002. Software complexity: Toward a unified theory of coupling and cohesion. *Proceedings of the ICSC Workshop*.
- Dasgupta, P., P.A. David. 1994. Towards a new economics of science. *Research Policy* **23** 487-524.
- Demsetz, H. 1964. The Welfare and Empirical Implications of Monopolistic Competition. *The Economic Journal* **74**(295) 623-641.
- Downs Jr, G., L. Mohr. 1976. Conceptual Issues in the Study of Innovation. *Administrative Science Quarterly*.
- Emanuelsson, P., U. Nilsson. 2008. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science* **217** 5-21.
- Feller, J., B. Fitzgerald, S. Hissam, K.R. Lakhani, eds. 2005. *Perspectives on Free and Open Source Software*. MIT Press, Cambridge.
- Fichman, R., C. Kemerer. 2001. Incentive compatibility and systematic software reuse. *The Journal of Systems & Software*.
- Fichman, R.G., D.F. Kemerer. 1997. The assimilation of software process innovations: An organizational learning perspective. *Management Science* **43**(10) 1345-1363.
- Fleming, L. 2001. Recombinant Uncertainty in Technological Search. *Management Science* **47**(1) 117-132.
- Gill, G.K., C.F. Kemerer. 1991. Cyclomatic Complexity Density and Software Maintenance Productivity. *IEEE Transactions on Software Engineering* **17**(12) 1284-1288.
- Gulley, N. 2004. In praise of tweaking: a wiki-like programming contest. *ACM Interactions* **11**(3) 18-23.
- Haefliger, S., G. Von Krogh, S. Spaeth. 2008. Code Reuse in Open Source Software. *Management Science* **54**(1) 180-193.

- Hess, C., E. Ostrom. 2003. Ideas, Artifacts, and Facilities: Information as a Common-Pool Resource. *Law and Contemporary Problems* **66** (12) 111-145.
- Hertel, G., S. Niedner, S. Herrmann. 2003. Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel. *Research Policy* **32**(7) 1159-1177.
- Höltkä, K., E. Suh, O. de Weck. 2005. Tradeoff between modularity and performance for engineered systems and products. *ICED 2005: The 15th International Conference on Engineering Desing.*
- Johnson, S. 1978. Lint, a C program checker. *Computer Science Technical Report.*
- Katila, R., G. Ahuja. 2002. Something Old, Something New: A Longitudinal Study of Search Behavior and New Product Introduction. *Academy of Management Journal* **45**(6) 1183-1194.
- Kendall, G., A. Parkes, K. Spoerer. 2008. A Survey of NP-Complete Puzzles. *ICGA Journal* **31**(1) 13-35.
- Knight, J., M. Dunn. 1998. Software quality through domain-; driven certification. *Annals of Software Engineering* **5**(1) 293-315.
- Lakhani, K.R., E. von Hippel. 2003. How Open Source Software Works: Free User to User Assistance. *Research Policy* **32**(6) 923-943.
- Lakhani, K.R., R. Wolf. 2005. Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects. J. Feller, B. Fitzgerald, S. Hissam, K.R. Lakhani, eds. *Perspectives on Free and Open Source Software.* MIT Press, Cambridge, MA.
- Lerner, J., J. Tirole. 2002. Some Simple Economics of Open Source. *Journal of Industrial Economics* **50**(2) 197-234.
- Levine, S. S., M. J. Prietula. 2006. Towards a Contingency Theory of Knowledge Exchange in Organizations. K.M. Weaver, ed. *Best Paper Proceedings.* Academy of Management, Atlanta, GA.
- Levine, S. S., M. J. Prietula. 2008. Towards a Theory of Collective Open Source Innovation. *Working Paper.*
- Lynex, A., P. Layzell. 1998. Organisational considerations for software reuse. *Annals of Software Engineering.*
- MacCormack, A., J. Rusnak, C.Y. Baldwin. 2006. Exploring the Structure of Complex Software Design: An Empirical Study of Open Source and Proprietary Code. *Management Science* **52**(7) 1015-1030.
- Majchrzak, A., L. Cooper, O. Neece. 2004. Knowledge Reuse for Innovation. *Management Science* **50**(2) 174-188.
- Markus, L. 2001. Towards a Theory of Knowledge Reuse: Types of Knowledge Reuse Situations and Factors in Reuse Success. *Journal of Management Information Systems* **18**(1) 57-93.
- McCabe, T. 1976. A complexity measure. *IEEE Transactions on Software Engineering* **SE-2**(4) 308-320.
- Merton, R.K. 1973. The Normative Structure of Science. N.W. Storer, ed. *The Sociology of Science.* The University of Chicago Press Chicago, 267-278.
- Meyers, S., M. Lejter. 1991. Automatic Detection of C++ Programming Errors: Initial thoughts on a lint. Brown University Computer Science Department, Providence, RI.
- Mokyr, J. 2002. *The Gifts of Athena: Historical Origins of the Knowledge Economy.* Princeton University Press, Princeton, NJ.
- Murray, F., S. O'Mahony. 2007. Exploring the Foundations of Cumulative Innovation: Implications for Organization Science. *Organization Science* **18**(2) 1006-1021.
- Nuvolari, A. 2004. Collective invention during the British Industrial Revolution: the case of the Cornish pumping engine. *Cambridge Journal of Economics* **28**(3) 347-363.
- Osterloh, M., S. Rota. 2007. Open source software development—Just another case of collective invention? *Research Policy.*
- Poulin, J. 1995. Populating software repositories: Incentives and domain-specific software. *The Journal of Systems & Software* **30**(3) 187-199.
- Raymond, E. 1999. *The Cathedral and the Bazaar: Musings on Linux and Open Source from an Accidental Revolutionary.* O'Reilly and Associates, Sebastapol: CA.
- Roberts, J., I.-H. Hann, S. Slaughter. 2006. Understanding the Motivations, Participation, and Performance of Open Source Software Developers: A Longitudinal Study of the Apache Projects. *Management Science* **52**(7) 984-999.
- Rothenberger, M., K. Dooley, U. Kulkarni, N. Nada. 2003. Strategies for software reuse: A principal component analysis of reuse practices. *IEEE Transactions on Software Engineering* **29**(9) 825-837.

- Ryan, R.M., E.L. Deci. 2000. Intrinsic and Extrinsic Motivations: Classic Definitions and New Directions. *Contemporary Educational Psychology* **25** 54-67.
- Salus, P.H. 1994. *A Quarter Century of UNIX*. Addison-Wesley, Reading, MA.
- Schumpeter, J.A. 1934. *The Theory of Economic Development: An Inquiry into Profits, Capital, Credit, Interest and the Business Cycle*. Oxford University Press, London.
- Scotchmer, S. 1991. Standing on the Shoulders of Giants: Cumulative Research and the Patent Law. *The Journal of Economic Perspectives* **5**(1) 14.
- Simcoe, T. 2007. XTPQML: Stata module to estimate Fixed-effects Poisson (Quasi-ML) regression with robust standard errors. Available at <http://ideas.repec.org/c/boc/bocode/s456821.html>
- Sojer, M., J. Henkel. 2009. Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments. Available at SSRN: <http://ssrn.com/abstract=1489789>
- Sorensen, J.B., T.E. Stuart. 2000. Aging, Obsolescence, and Organizational Innovation. *Administrative Science Quarterly* **45**(1) 81-112.
- Stallman, R. 2001. The Free Software Definition. Free Software Foundation.
- Stephan, P.E., S.G. Levin. 1992. *Striking the Mother Lode in Science: The Importance of Age, Place, and Time*. Oxford University Press, New York.
- Stewart, K., S. Gosain. 2006. The Impact of Ideology on Effectiveness in Open Source Software Development Teams. *MIS Quarterly* **30**(2) 291-314.
- Stuermer, M., S. Spaeth, G. Von Krogh. 2009. Extending Private-Collective Innovation: A Case Study. *R&D Management* **39**(2) 170-191.
- von Hippel, E. 2001. Innovation by User Communities: Learning from Open Source Software. *Sloan Management Review* **42**(4) 82-86.
- von Hippel, E., G. von Krogh. 2003. Open Source Software and the Private-Collective Innovation Model: Issues for Organization Science. *Organization Science* **14**(12) 209-223.
- Weber, S. 2004. *The Success of Open Source*. Harvard University Press, Cambridge, MA.
- Wood, R.E. 1996. Task complexity: Definition of the construct. *Organizational Behavior and Human Decision Processes* **37**(1) 60-82.
- Wooldridge, J. 1999. Distribution-free estimation of some nonlinear panel data models. *Journal of Econometrics* **90**(1) 77-97.
- Zuse, H., P. Bollmann. 1989. Software Metrics: Using measurement theory to describe the properties and scales of static software complexity metrics. *ACM SIGPLAN Notices* **24**(8).

Figure 1 - Histogram of Submissions

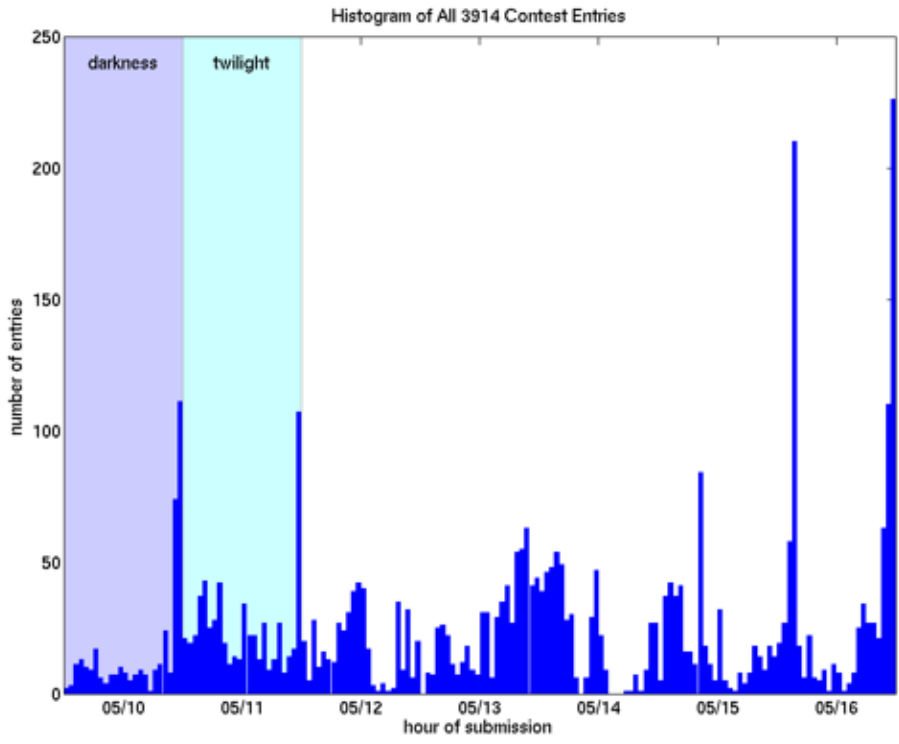
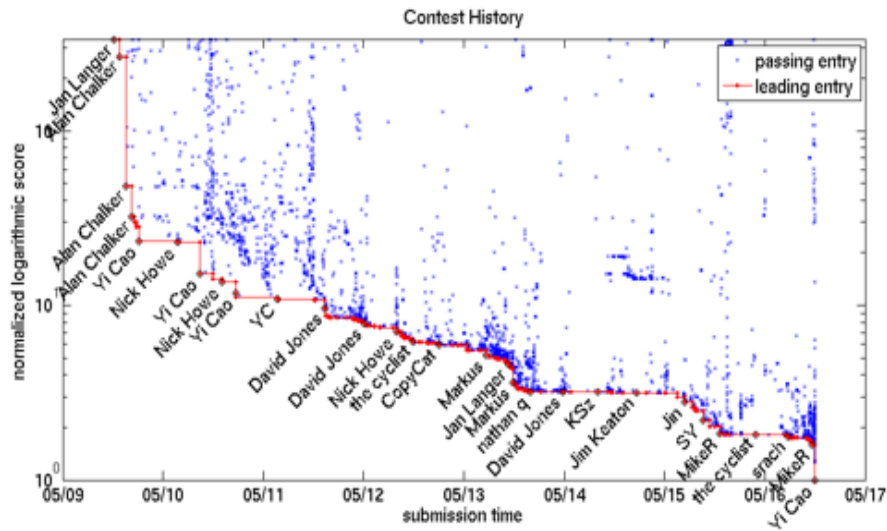


Figure 2 – Distribution of Scores in a Contest



Notes:

- 1 - Y-axis is on a normalized logarithmic scale with last best entry at score = 1.
- 2 - Red (Dark) line marks entries that were top performing at any given time.

Figure 3: Conceptual Model and Study Hypotheses

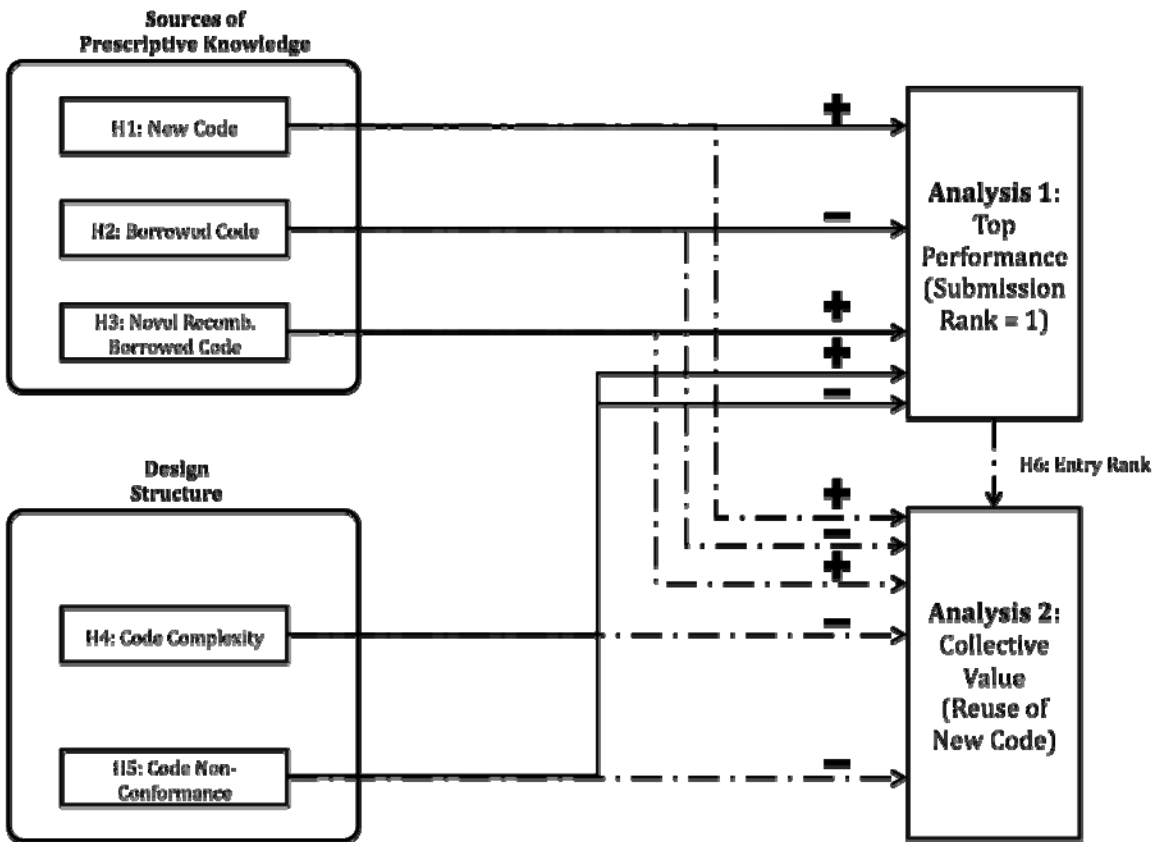


Figure 4: Summary of Findings Only Significant Coefficients Reported

Hypotheses	Top Rank	Collective Value	Collective Value – First Adoption	Collective Value – Lead Entry Adoption
Sources of Knowledge				
H1: New Code	0.421***	0.392***	0.377***	0.329***
H2: Borrowed Code	-0.125***	0.297*	0.319***	n.s.
H3: Novel Recombination of Borrowed Code	0.116***	0.073**	0.048***	n.s.
Design Structure				
H4: Code Complexity	1.600***	0.254*	0.155***	n.s.
H5: Code Non-Conformance	0.146*	-0.414***	-0.142*	-0.388***
Performance				
H6: Entry Rank	n.a.	-0.269***	-0.173***	-0.239***

Notes:

Coefficient estimate should be read as percentage change in the odds of achieving the top rank or the percentage change in the count of new lines reused for collective value – given a one percentage change in the independent variable of interest.

n.s.: Not significant.

n.a.: Not applicable

Table 1: Participation Patterns across All Contests

	Date Held	Number of submissions	Percentage of submissions that failed	Number of participants	Avg. submissions per participant	Avg. lines of code	Avg. novel lines of code	Avg. number of lines of code borrowed (from others)	Avg. pair-wise novel combinations of borrowed lines	Avg. number of people code borrowed from	Avg. McCabe Complexity	Avg. Lint count
					11	76	4	48	80	8	12	4
Molecule	5/02	1631	40%	153	(27)	(50)	(11)	(39)	(523)	(5)	(13)	(6)
					12	382	4	330	476	20	37	90
Protein Folding	11/02	2437	23%	199	(28)	(364)	(11)	(330)	(5314)	(14)	(29)	(134)
					11	265	4	222	295	19	14	14
Trucking Logistics	4/03	1659	18%	145	(29)	(159)	(14)	(148)	(4650)	(9)	(9)	(11)
					15	407	7	364	374	25	39	19
Election Gerrymander	4/04	2392	15%	157	(35)	(255)	(213)	(258)	(3057)	(15)	(27)	(22)
					20	718	5	643	1054	18	41	47
Furniture Moving	11/04	1834	31%	94	(42)	(443)	(21)	(448)	(9394)	(11)	(34)	(44)
					14	106	6	76	69	13	48	10
Ants	5/05	2206	11%	153	(38)	(45)	(18)	(49)	(874)	(5)	(36)	(20)
					18	197	6	162	133	16	25	9
Sudoku	11/05	3061	20%	168	(42)	(109)	(18)	(122)	(1350)	(10)	(19)	(18)
					26	120	5	89	54	13	18	9
Blockbuster	4/06	4420	17%	168	(111)	(47)	(15)	(57)	(493)	(7)	(65)	(11)
					28	227	4	202	94	16	65	23
Blackbox	11/06	4933	14%	175	(118)	(154)	(20)	(154)	(1719)	(9)	(51)	(29)
					34	221	4	184	99	15	8	7
Peg Jumping	5/07	3912	12%	116	(92)	(128)	(15)	(144)	(1271)	(9)	(6)	(7)
					25	250	18	190	174	15	13	12
Gene Splicing	11/07	3285	18%	132	(71)	(121)	(67)	(140)	(1829)	(9)	(8)	(16)
					19	250	6	211	215	16	30	21
Average Per Contest		2888 (1134)	20% (0.087)	150	(67)	(241)	(27)	(235)	(3229)	(10)	(33)	(48)

Table 2: Description and Summary Statistics of Dependent and Independent Variables

	Description	Mean	S.D.	Min	Max	LEAD _ENT RY	SOCIAL_ VALUE	NEW_C ODE	Borrowed _Code	Novel Combo_ Borrowed	Code_ Complexity	Code_Non Conform.	Entry_Ra nk
LEAD_ENTRY	DV1: Submission is in lead upon debut in contest	0.06	0.23	0	1	1							
SOCIAL_VALUE	DV2: Count of new lines of code reused by other participants	170.69	2215.77	0	154773	0.093	1						
NEW_CODE	IV: New lines of code in submission	5.22	28.05	0	534	0.034	0.119	1					
BORROWED_CODE	IV: Lines of code in submission borrowed from other participants	246.3	238.36	0	1272	0.047	-0.008	-0.108	1				
NOVELCOMBO_BORROWED	IV: Pair-wise count of borrowed lines of code not co-occurring in prior entries by anyone	247.85	3504.34	0	193126	0.002	0.019	0.015	0.089	1			
CODE_COMPLEXITY	IV: McCabe Complexity measure of submission	33.18	34.65	0	669	0.003	0.002	-0.053	0.256	0.006	1		
CODE_NON-CONFORMANCE	IV: Lint message count of submission	22.76	50.95	0	576	0.061	0.007	0.04	0.431	0.037	0.22	1	
ENTRY_RANK	IV: Rank at debut of submission	666.29	959.62	1	4934	-0.173	-0.03	0.011	-0.121	0.004	-0.169	-0.19	1

**Table 3: Logistic Regression that Predicts Whether Submission by Author Debuts at Rank One
(All Variables are Logged)**

	3-1		3-2		3-3		3-4		3-5		3-6		3-7	
NEW_CODE	0.447	(0.039)***											0.421	(0.042)***
BORROWED_CODE			-0.306	(0.048)***									-0.125	(0.050)***
NOVELCOMBO_ BORROWED					0.104	(0.021)***							0.116	(0.017)***
CODE_COMPLEXITY							0.076	(0.079)	1.744	(0.421)***			1.600	(0.417)***
CODE_COMPLEXITY_ SQUARED									-0.284	(0.071)***			-0.258	(0.070)***
CODE_NON- CONFORMANCE											0.181	(0.078)*	0.146	(0.068)*
CONSTANT	-1.884	(0.923)*	-1.879	(0.927)*	-1.120	(0.930)	-1.362	(0.940)	-2.832	(0.980)**	-0.614	(0.947)	-3.071	(0.932)***
Wald Chi Square	2026.646***		1892.886***		1790.08***		1786.675***		1746.488***		1866.411***		2081.705***	
Pseudo R Square	0.297		0.286		0.287		0.283		0.286		0.284		0.307	
BIC	8625.464		8750.899		8740.033		8791.697		8761.682		8773.415		8556.257	

Robust standard errors in ().

Standard errors are clustered by 1,306 unique authors.

Controls included but not reported: Contest (Dummy for specific contest), Time (Upon submission: total number of participants in contest, number of other submissions made in hour of submission, total time left in contest, hours elapsed in contest (integer value)), Author (number of previous submissions made in contest, number of previous submissions at rank = 1), Entry (code size, lines of code novel from current leader).

Number of observations: 26,927.

*p < 5%, **p < 1%, ***p < 0.1% significance level.

**Table 4: Quasi-Maximum Likelihood Fixed Effects Poisson Regression on Reuse of New Lines of Code in Submissions by Others
(All Variables are Logged)**

	Dependent Variable = Collective_Value								
	Dependent Variable= Coll._Value_First			Dependent Variable= Coll._Value_Lead					
	4-1	4-2	4-3	4-4	4-5	4-6	4-7	4-8	4-9
ENTRY_RANK	-0.286 (0.089)***						-0.269 (0.064)***	-0.173 (0.028)***	-0.239 (0.049)***
NEW_CODE		0.169 (0.148)					0.392 (0.114)***	0.377 (0.060)***	0.329 (0.100)***
BORROWED_CODE			0.273 (0.126)*				0.297 (0.037)*	0.319 (0.067)***	0.102 (0.109)
NOVELCOMBO_ BORROWED				0.094 (0.0416)*			0.073 (0.026)**	0.048 (0.012)***	0.053 (0.028)
CODE_COMPLEXITY					0.161 (0.080)*		0.254 (0.010)*	0.155 (0.056)**	0.161 (0.097)
CODE_NON- CONFORMANCE						-0.345 (0.065)***	-0.414 (0.074)***	-0.142 (0.058)*	-0.388 (0.076)***
Wald Chi Square	3551390***	66437.5***	2322.151***	80815.96***	41758.47***	30278***	1.15E+15***	1.01E+16***	8.24E+13***

Robust standard errors in ().

Standard errors are clustered by author in contest.

Controls included but not reported: Contest(Dummy for specific contest), Time(Upon submission: number of total participants in contest, number of other submissions made in hour of submission, total time left in contest, hours elapsed in contest (integer value)), Author (number of previous submissions made in contest, number of previous submissions at rank = 1), Entry(code size, lines of code novel from current leader, reuse of new lines of code by author until at least one line was part of a new submission from author or other that had rank =1).

Number of Observations: 26,927.

*p < 5%, **p < 1%, ***p < 0.1% significance level.