Methods as Theories: Evidence and Arguments for Theorizing on Software Development

Steve Sawyer
School of Information Sciences and Technology
The Pennsylvania State University
sawyer@ist.psu.edu

Hala Annabi
The Information School
University of Washington
hpannabi@u.washington.edu

A modified version of this paper is to be published as: Sawyer, S. and Annabi, H. (2006) "Methods as Theories: Evidence and Arguments for Theorizing on Software Development," In Trauth, E , Howcraft, D. and DeGross, J. (Eds.) *Social Inclusion in the Information Society*, London, Kluwer, in press.

Abstract

In this paper we argue that software development methods represent theories on how best to engage the impressively complex and inherently socio-technical activity of making software. To help illustrate our points we draw on examples of three software methods: the waterfall approach, packages software development and free/libre and open source software development, In doing this, we highlight that software development methods reflect – too often implicitly – theories of (1) how people should behave, (2) how groups of people should interact, (3) the tasks that people should do, (4) the order of these tasks, (5) the tools needed to achieve these tasks, (6) the proper outcomes of these tasks, (7) the means to make this all happen, and, (8) that these relations among concepts are further set in specific contexts. We conclude by highlighting three trends in conceptualizing these eight elements.

Methods as Theories: Evidence and Arguments for Theorizing on Software Development

Through this paper we argue that software development methods are vehicles for theorizing on the impressively complex and inherently socio-technical activity of making software work. We support this argument by comparing the systems development lifecycle with packaged software development and Free/Libre and open source methods of software development.

Framing software development as socio-technical is common (e.g. Sawyer 2004; Scacchi 2002). A socio-technical lens demands we explicitly attend to the bound-up nature of people, particular technological elements, and the contexts of these nuanced inter-dependencies (Bijker, 1995; Bijker, Hughes and Pinch, 1987; Law and Bijker, 1992)[1]. Social aspects of software development include how people interact, behave and organize. Technological aspects of software development include the use of methods, techniques and computing technologies. In practice it is difficult to disentangle the ways people do things from the methods, techniques and computing technologies they use for this doing.

We further note that software development differs from information systems (IS) development methods in at least two ways (Sawyer, 2001). First, software development is focused on the development of an artifact – some defined set of working code that reflects specifications. An IS development effort is further focused on ensuring that software is brought together with specific users in specific organizational settings.

Second, current trends in labor specialization are reflected in differences among skill sets of those that develop software and those that implement IS. Simply, software engineers do different work than information systems consultants. And, these two groups of people tend to work for different organizations, separate from one-another and from the consumer organization (who purchased the software but needs an IS). This division of labor is made more clear by considering the business analysts, trainers, technical specialists, usability staff, and others who serve to make an IS from software.

The range of new approaches to both software and IS development, the constant evolution in current approaches, and the ongoing attention suggest that our theorizing is still incomplete. Here we focus specifically on the underlying, and too-often implicit, elements of this theorizing. We begin here arguing that all software development methods have eight common elements. To support this argument we draw on a comparison of three software development approaches. We conclude with specific suggestions for improving our theorizing on methods.

SOFTWARE DEVELOPMENT AS THEORIZING

We contend that software development methods are a form of theory[2]. A theory is a relationship between (or among) two (or more) concepts (Merton, 1967). The simplest form is causal: "A"

[1] This conceptual perspective is increasingly known as 'social informatics.' A simple summary of social informatics is presented in Sawyer (2005). More complete discussions can be found in Kling, Rosenbaum and Sawyer (2005); Kling (1999) and Kling (2000).

[2] In this paper we make the claim and illustrate our position, leaving to other papers the conceptual justification. As such, the value of this paper is the support we can provide this premise.

leads to "B." An example would be to be born (A) leads to, at some point, dieing (B). Typically the goal of theory is to develop the relationships among the constructs to such a point that the theory is general, specific, and accurate. These three criteria, however, are not mutually attainable. The principle known as "Occam's Razor" suggests that any theory can pursue two of three criteria, sacrificing the third. In our example, we pursued accuracy and generality, but could not pursue specificity.

Developing, testing and using theory are central goals in contemporary scholarship. Variations in representation, notation, value of directionality, and centrality of theory are contentious topics in every academic discipline. We acknowledge the passion this topic engenders, leaving to others a more detailed engagement. Our point in raising theory as a scholarly goal is to aver that in software development, methods are forms of theory (they identify concepts and presuppose relations). Weick (1995) makes clear that this is a process which he calls theorizing. He goes on to argue the importance of making theorizing more visible to scholars and others. We agree and in this paper do so by examining the common conceptual elements that make up software development methods-as-theory[3].

Socio-technical theorizing

Our efforts to theorize software development as a socio-technical activity builds on the social-shaping of technology (SST) perspectives developed in Bijker (1995), Law and Bijker, (1992) and Bijker, et. al., (1987). The SST perspective highlights that the material characteristics and actions of any technology are shaped by the social actions of the designers, the specific uses of that technology and the evolving patterns of use over time. This differs from the co-design approach that is prevalent in North America. As Scacchi (2005) notes, the co-design approach too-often evolves into a benign neglect of the interaction between what is social and technical, leading to an evocation of the concepts without a concomitant analytical activity.

Bijker's (1995) four socio-technical principles frame our theorizing. The *seamless web* principle states that any socio-technical analysis should not *a priori* privilege technological or material explanations ahead of social explanations, and vice versa. The principle of *change and continuity* argues that socio-technical analyses must account for both change and continuity, not just one or the other. The *symmetry principle* states that the successful working of a technology must be explained as a process, rather than assumed to be the outcome of 'superior technology'. The *actor and structure principle* states that socio-technical analyses should address both the actor-oriented side of social behavior, with its actor strategies and micro interactions, and structure-oriented side of social behavior, with its larger collective and institutionalized social norms and processes.

Software development methods as sociotechnical theories

Two elements of any software development method-as-theory are guidance on how people should behave and how groups of people should interact. For example, one might theorize that people are to share information selflessly and pursue ego-less programming. Another approach

[3] In this paper we take the first step of engaging the concepts that are part of software development methods as theory, leaving to future work the task of sorting through the relationships among these concepts.

would be to theorize that people will have differences of opinion and that there will be inter-personal conflict among team members.

Software development methods also reflect a set of expectation relative to the tasks that people should do, the order of these tasks, and the tools needed to achieve these tasks. The details of conduct, input and output elements, sequencing, and resource needs of tasks are central elements of any software development method and the core of this discourse (e.g. Egyedi, 2004; Cubranic & Booth, 1999).

The proper outcomes of these tasks continue to be an active area of scholarship. On trend is a steady movement towards multiple measures and to focusing on measure of use in addition to measures of the software artifact's structure, size and technical performance (Melone, 1990). The rise of open source development has elevated the attention to theorizing on the reasons why developers perform, and the incentives that encourage (and discourage) performance (e.g. Bergquist & Ljungberg, 2001).

Software development methods also incorporate, explicitly or implicitly, relations to specific contexts. For example, clean-room, participatory, and packaged software development approaches demand separation, inclusion, or distance from the user. The literature on virtual teams and distributed development make clear a second form of context (geography) (e.g. Moon & Sproull, 2000). The differences among custom and packaged software further suggest that the industrial environment matters (e.g., Sawyer, 2000).

In sum, as laid out in Table 1, software development methods are explicit representations of: (1) how people should behave, (2) how groups of people should interact, (3) the tasks that people should do, (4) the order of these tasks, (5) the tools needed to achieve these tasks, (6) the proper outcomes of these tasks (including means and ways to evaluate these outcomes) and (7) the means to make this all happen. The relations among these concepts are further set in (8) specific contexts, implying that the exact nature of such relations are contingent to some degree on the larger social milieu.

Table 1: Concepts Important to Theorizing Software Development Methods

Element	Details
People's individual behavior	What is expected of people engaged in developing software
People's collective action	The interactions among people working together to develop software
Task selection	The particular tasks that need to be done to develop software
Task ordering	The ordering of the particular tasks to develop software
Tool support	The roles and featured of tools used to support tasks/ordering
Outcomes (measures)	The elements measured to assess both progress and completion
Incentives	The structures put in place to encourage positive, and discourage negative, behaviors and interactions
Contexts	The larger social, cultural, economic and industrial milieu in which software development takes place

COMPARING SDLC, PACKAGE AND FREE/LIBRE OPEN SOURCE SOFTWARE APPROACHES

To illustrate how these eight concepts underscore methods, we compare three approaches to developing software. We select the SDLC, packaged software development and FLOSS methods, providing in this section a brief review of these three approaches before developing our comparison (summarized in Table 2 and continued below).

Systems development life cycle

The systems development life cycle (SDLC) or waterfall approach is well-known, oft-referenced, and rarely followed. In the SDLC, specific steps are linearly sequenced with some overlap between steps to allow for knowledge transfer. Specific skills and resources for each step, its inputs and outputs, and proper approaches to pursuing the transfer of inputs to outputs are documented.

The premise of the SDLC is that process drives outcomes. The measures for SDLC success typically include cost, quality, user satisfaction as recognition of value to the larger corporate mission. Implicit in the SDLC are at least two relevant assumptions. First, it implies that software development takes place within one organization (or, at least, is totally controlled by that organization – i.e., when hiring a contractor/consultant to construct a custom product). This reflects vertical integration (a hierarchy). Second, the SDLC is focused on building, not buying, software. This is appropriate, for that was its purpose. .

Staff costs should be minimized and typical SDLC-based efforts are characterized by team membership turnover, division of labor by both phase and function, and disbanding following the completion of the first release (Cusumano and Smith, 1997). Thus, these are more like ad-hoc work groups, not teams (Goodman, Ravlin and Argote, 1986).

Packaged software development

Packaged software (also known as shrink-wrapped, commercial-off-the-shelf (COTS) and commercial software) means the code is sold as a (licensed) product (purchased from a vendor, distributor or store) for all computer platforms including mainframes, work-stations and microcomputers (Carmel, 1997; Carmel and Sawyer, 1998; Sawyer, 2000).

In PSD time pressures (not cost) drive development. Packaged software developers tend to have a product (not process) view of development (Carmel, 1995; Carmel and Becker, 1995; Cusumano and Smith, 1997). A product focus means that the dominant goal of the software development effort is to ship a product. This product focus also implies that these products have distinct trajectories with the software evolving through a planned set of releases.

In packaged software firms, developers hold line positions so their needs are central to the performance of the organization. In effect, they are the company's production mechanism as they generate revenue. Packaged software developers often have at best a distant relationship with their user population. This separation means that intermediaries – such as help desk

personnel and consultants – link users to developers (Maiden and Ncube, 1998; Keil and Carmel, 1995; Grudin, 1991).

Packaged software products are measured by criteria such as favorable product reviews in trade publications, the degree of "mind share" – the awareness of a product in the minds of the target population, developing a large installed base and/or creating new markets (Brynjolfsson, 1994; Andersson and Nilsson, 1996).

Free/Libre open source software

FLOSS is a broad categorization used to describe software developed and released under various "open source" licenses. Licenses offer a range of features, all allowing inspection of the software's source code. We use the term FLOSS to encompass the Free Software movement, which also releases software along the same terms as the OSS movement, but with a distinction that derivative works must be made available under the same non-restrictive license terms. FLOSS projects comprise of a varying number of developers ranging from a few to a hundred or more. FLOSS development groups are groups working in distributed computer mediated networked form (Scacchi, 2002).

FLOSS members interact primarily or exclusively via computer-mediated communications (CMC). Project members coordinate their activities primarily through private e-mail, mailing lists, bulletin boards and chat room and use compilers, bug tracking and version control systems for their software development.

In general, FLOSS processes are fluid not complying with any particular software engineering method (Raymond, 1998; Scacchi, 2002). One of the most commonly mentioned models used to explain the methods or practices of FLOSS development is Raymond's (1998) "The cathedral and the bazaar" metaphor. Raymond depicts FLOSS developers as autonomously deciding schedule and contribution modes for software development as merchants in a bazaar would, thereby dismissing the need for central coordination as the construction of a cathedral would in a master architect. The bazaar metaphor is limited as it diminishes aspects of the FLOSS development process, such as the role of the project leader or core group and the existence of de-facto hierarchies (Bezroukov, 1999).

In FLOSS a mixture of self-serving and altruistic goals drive development. Developers join FLOSS projects for one or more of several reasons, some of which are employment (as some FLOSS developers are employed by formal organizations to develop software), to meet a personal need, to contribute to creating a public good, to gain satisfaction from the software development process, and/or for potential career gains (Moody, 2001).

Members of any project move from peripheral roles to a core developer role in the project through a merit base process (Cubranic and Booth, 1999). An individual's technical expertise and participation in developing the product results in his/her inclusion in the core group of developers. However, to become one of the core developers means they must have a detailed understanding of the software and development processes. Since there is no separate documentation for system requirements or design, this poses a significant barrier to entry

(Fielding, 1997; Hecker, 1999). Designs and requirements evolve over time and are implicitly articulated in public mailing lists as a result of individual developers' desired functionality and a developer's willingness to implement them (Scacchi, 2002). Tasks are accomplished based on developers' needs and interests and articulated in to do lists as seen in Apache Web Server in the early years (Annabi, 2005).

FLOSS software success is measured by a variety of criteria. User satisfaction, portability, favorable product reviews, learning opportunities, user-base, developer satisfaction and developer recognition are some of the measure of FLOSS success (Crowston, Annabi and Howison, 2003). Portfolios of measures can be used to assess any particular project depending on project and members goals.

Methods as theories: Comparing the three approaches

In Table 2 and below we highlight via comparisons how the eight elements of a software development method reflect theorizing. A complete analysis is beyond the scope of this paper; so, here we summarize the concepts. Here we note the presence of these concepts in each of the three approaches, leaving to other work attention to relationships among the concepts.

People's individual behavior. Each of the three approaches make clear expectations for a certain set of behaviors from people. In the SDLC, people are to attend to process, share information, suppress ego issues, and focus on developing role-specific technical and professional skills. The PSD approach conceives of people as vproduct-focused, competitive, technically skills, and with limited need for social skills, willing to take risks, and time-pressured. In FLOSS, people are seen as pursuing a mix of altruistic and self-serving goals, constrained by social controls, and with high technical skill levels.

People's collective action. In the SDLC, people are expected to be oriented to the goals of the collective and consensus is expected. In PSD, people's interactions will be guided by product needs, time pressures, and profit, and conflict is expected. In FLOSS, people's collective behavior is guided by the twin goals of public good and personal needs, and interactions are driven by performance goals.

Task selection. In the SDLC, tasks are pre-defined by the method and system requirements – an engineering ethic. The task inputs, outputs and means of proceeding are specified and often inflexible. In the PSD approach, tasks are more flexible, though there are common templates or forms that must be met. In FLOSS, tasks are mostly left to the developers, with a few (such as version control) serving as central aspects of the effort. These tasks help to structure FLOSS.

Task ordering. In the SDLC, task ordering is typically fixed, linear, and prescribed. In PSD, the ordering of tasks is more fluid while particular inputs and outputs are less prescribed. It is difficult to develop a task ordering in FLOSS beyond observing that certain tasks (such as the use of a configuration management tool) serve as central and structuring elements of the approach.

Tool support. In the SDLC, software tools are used to enforce task ordering and task completion (a controlled production environment). These tools are often integrated, complex and have limited flexibility. In both the PSD and FLOSS approaches, tools are engaged that support collaboration, coordination and production support. These tools are more flexible, less integrated, and often quite simple.

Table 2: Comparing SDLC, Package and FLOSS Development Approaches

Element	SDLC	Package	Open Source
People's individual behavior	Process-focused, Specialized to particular roles, Sequenced, ego-less, and sharing-oriented	Product-focused, Competitive, skill-based, interdependent, time-pressured.	Self interest, skill-focused and altruistic.
People's collective action	Collective, controlled and focused on error reduction	Conflictual, focused on delivery, and coding.	Product focused, focused on personal goals *and* producing a public good
Task selection	Defined by system requirements and mostly inflexible	Group defined and mostly flexible	Driven by self interest and supported by a merit system
Task ordering	Prescribed by phase or function	Iterative	Fluid, flexible, often iterative
Tool support	Enforce control and process adherence	Support interactions and interdependencies	Support interaction and sharing code
Outcomes (measures)	Adoption, customer satisfaction, cost	Market share, user and industry review	Developer satisfaction, market share, reviews, portability
Incentives	Income, skill-development	Profit, recognition	Developer personal satisfaction, project and developer recognition, public good,
Contexts	Organizational	Market driven	User-base
Notes	Users involved through intermediaries	Users involved through intermediaries	Users directly involved (developers are often users, and users test and fix bugs and contribute code)

Outcomes (measures). In the SDLC, process measures are used, and these measures focus on cost, quality, and user take-up/value. In PSD, product measures are used, and these measures focus on installed base/market share, sales, margin, and defect rates. In FLOSS, there is a combination of product and personal measures. This remains an active area of inquiry and continues to be poorly understood (see Crowston, Annabi and Howison, 2003).

Incentives. In the SDLC, developers' behaviors are motivated by salary/income, since developers are employed, as well as opportunities to learn new skills from developing particular software. In PSD, incentives extend past salary to include stock options and shares of sales. In FLOSS, there are a variety of incentives depending on circumstances, but generally, developers are interested in developing a product for the common good while meeting their own needs, attaining satisfaction from engaging in the development process and potentially gaining recognition for themselves and the project.

Contexts. In the SDLC, software development occurs in the context of organizational goals, needs and capabilities. The main purpose is to meet organizational objectives within limited budgets while accommodating social, technological, and political factors. In the SDLC, risk mitigation is a central issue. In PSD, software is developed to meet a need present or forecasted in the market, and to pursue opportunities. In PSD, taking on risk is a central issue. In FLOSS, software is developed by users for users as ideas about the product evolved over time with user influence being the main driver. In FLOSS, risk is borne primarily by individuals.

We further note that the roles that users play shapes development. And, this shaping has both direct and indirect component. For example, in the SDLC, users are the focus of one phase, and then kept distant from the development effort. However, a focus on meeting user's needs dominates the SDLC approach. In the PSD, users are always distant, but there are extensive efforts to gather user needs and monitor their interests. These efforts, however, are one of several factors that influence design. In FLOSS, developers are often users, and the blurry boundary between users and developers creates an interesting dynamic for development.

Observations

Drawing on this analysis, in Table 3 and below we summarize and discuss eight observations regarding software development methods as incipient theories.

Table 3: Observations on Theorizing Software Development Methods

Element	Details
People's individual behavior	An increasingly richer view of people as having passion, engaging conflict and pursuing personal agendas (not just as error-producing and limited cognitive agents).
People's collective action	Interactions among people are central characteristics of methods and must be accounted for in the design of tasks, tools and outcome measures.
Task selection	Tasks are becoming more fluid and more flexible.
Task ordering	Task ordering is becoming less linear
Tool support	Tool support moving towards supporting interaction and access to materials, not ('just') code production and process enforcement.
Outcomes (measures)	Measures are expanding and evolving.
Incentives	Incentives are under-explored (though FLOSS approaches require engaging this directly)

Contexts	Contexts are under-explored (though evidence and awareness that a one-size-fits-all approach to software development is growing).
Notes	The number, needs, skills, social power and other resources of users have substantial and multiple, indirect, effects on how software is developed.

People's individual behavior. We observe a trend towards an increasingly more complex view of people as having passion, engaging conflict and pursuing personal agendas (not just as error-producing and limited cognitive agents). This is most evident in FLOSS development as one of the top reasons and features of the development process is "satisfying an itch" for developing and creating to meet needs (Crowston, Annabi and Howison, 2003).

People's collective action. We note that, there is a shift towards conceiving the interaction among people eas central characteristics of methods that must be accounted for in the design of tasks, tools and outcome measures. Both the PSD and FLOSS literature makes clear that managing interactions are central issues to success (Sawyer, 2000; Annabi, 2005; Crowston, Annabi and Howison, 2003).

Task selection. We observe that tasks structures are seen as more fluid and responsive. Tasks are defined and chosen through consensus and conflict and with user involvement. As in the cases of both PSD and FLOSS, developers define task through interactions categorized by both conflict and consensus.

Task ordering. We observe that task ordering is becoming more iterative, with expectations among developers and users that specific tasks are likely to have multiples passes. A second aspect of this iterative orientation is the increased flexibility in the order of tasks.

Tool support. We observe the roles and uses of software development tools are moving towards supporting interaction and access to materials. In doing this the tools are moving away from focusing solely on code production and process enforcement. The most vivid example of this is the FLOSS uses of version control software (Shaikh and Cornford, 2004)

Measures. The number of measures being used to evaluate software development continues to expand. These measures can be seen as a suite and encompass developer behavior, development team processes, measures of use and value to customers, and measures of the artifacts size, quality and resources.

Incentives. It appears that incentives (and disincentives) remain an under-explored area in SDLC and PSD. FLOSS development suggests that incentives are both intrinsic and extrinsic (Crowston, Annabi and Howison, 2003) affecting developers' interactions with the product and others in the development group.

Contexts. We observe that the professional community of software developers, and many in the academic community studying software development, are aware that there is no one-size-fits-all approach to software development. There are, however, common elements that define software

development (the point we are arguing here …) and that the way these elements are engaged is driven in part by the context. This contingency perspective suggests that differences in software development methods are critical.

We note that for more than 20 years, scholars have noted that the number, needs, skills, social power and other resources of users have substantial and multiple effects on how software is developed (e.g., Kling and Iacona, ; Markus, 1983; Keil and Carmel, 1995). It appears this user pressure is influencing the recent work in software development methods. We further note that the variations among these concepts across the three examples suggests that while the concepts are common, the pattern of relationships among these concepts differ.

IMPLICATIONS

We have argued here that software development methods can be best understood as theories, posited that these engage eight concepts. Drawing on a comparison of three approaches, we have observed that users and uses help to shape software development in many direct and indirect ways. In Table 4 and below, we look beyond our current observations to speculate on how we might advance theorizing on software development methods.

Table 4: Guidance for continued theorizing on software development methods

Leverage Points:
1. More complex representations of people's behaviors
2. More fluid task elements supported with more flexible tools
3. Increased integration of incentives, measures and context

More complex representations of people's behaviors

We expect that future software development methods will have more nuanced and complex representations of people's behaviors. For example, we expect that developers will be increasingly construed as problem-solvers (and not error-prone code writers) (Mockus & Herbsleb, 2002).. In part this more complex view of software developers is driven by our increased understanding of their work. It may also, at least indirectly, draw on our increased understanding of knowledge-based work. That is, we see software developers as knowledge workers, and increasingly they are able to choose what projects to join (Drucker, 1998; Annabi, 2005). This more nuanced view of people's behaviors is likely to drive a resurgence of empirical studies on performance and process (such as is seen in the FLOSS literature).

A second trend we expect to see in future theorizing on software development is that the interactions among people, and among people and the various tools and repositors used in developing software, will be seen as a central activity (e.g., Mockus and Herbsleb, 2002; Scacchi, 2002b). As we note below, this will influence the design and uses of tools, and incentives. This trend is likely to be instantiated in guidance for pair programming, team development, structured communication, shared work environments and a more discourse-oriented approach to documenting decisions.

More fluid task elements supported with more flexible tools

We speculate that future theorizing on software development methods will build on the concepts of templates. A template-oriented view makes clear that these structures are a guide, to be interpreted, not followed. In contrast, the SDLC and other recipe-based views makes guidance more like scripts: inflexible and increasingly unwieldy (as exceptions and errors lead to expanded scripting). The move towards templates means a blurring of tasks and ordering (even, though the focus on particular inputs and outputs will sharpen). So, even as the sequence of tasks becomes more fluid, and perhaps less linear, the specific needs at templated points will increasingly become more clear (and better understood). And, as noted above and in Sawyer (2004), we speculate that the tools used will better support people's collaboration and interaction – going beyond production and control functions (e.g, Vessey and Sravanapudi, 1995).

Increased integration of incentives, measures and context

Future theorizing on software development methods will better align participant's incentives with tasks and in doing this, these incentives will reflect the more creative, problem-solving, collaborative nature of people (Halloran and Scherlis, 2002; Scacchi, 2005; Mockus and Herbsleb, 2005). These incentives are likely to draw on multiple measures and a better understanding of tradeoffs (e.g. for FLOSS portfolios of measures suggested by Crowston, Annabi, Howison 2003). And, these methods will reflect contingencies such as contextual pressures and needs, including users engagement issues.

Looking beyond these specific speculations regarding future theorizing on software development methods, we argue that such work will explicitly or implicitly engage the relationships among eight core concepts. More subtly, but perhaps more profoundly, we introduced three socio-technical principles as the basis of this theorizing. These principles engage us to consider people's actions as co-equals with tasks, to focus on processes as flexible and contextual, and to highlight both the structural and agentic nature of people and tools. If one takes seriously our position – that software development methods are incipient theories, then the socio-technical principles of theorizing provide the conceptual guidance for how to proceed.

REFERENCES

Andersson R. and Nilsson A (1996) The standard application package market – an industry in transition?. In *Advancing Your Business: People and Information Systems in Concert* (Lundeberg M and Sundgren B, Eds.). EFI: Stockholm School of Economics, Stockholm,Sweden.

Annabi, H. (2005). Moving from Individual Contribution to Group Learning: The Early Years of the Apache Web Server. Unpublished Ph.D. Dissertation, Syracuse University, Syracuse, NY.

Bendifallah, S. & Scacchi, W, (1989) Work Structures and Shifts: An Empirical Analysis of Software Specification Teamwork , *Proc. 11th. Intern. Conf. Software Engineering*, May, IEEE Computer Society Press, Pittsburgh, PA. 260-270.

Bergquist, M., & Ljungberg, J. (2001). The power of gifts: organizing social relationships in open source communities. *Information Systems Journal, 11*(4), 305-320.

Bezroukov, N. (1999). A second look at the Cathedral and the Bazaar. *First Monday, 4*(12).

Bijker, W. (1995). *Of Bicycles, Bakelites, and Bulbs: Toward a Theory of Socio-technical Change*. Cambridge, MA: MIT Press.

Bijker, W., Hughes, T. & Pinch, T. (1987). *The Social Construction of Technological Systems*. Cambridge, MA: MIT Press.

Brynjolfsson E (1994) The productivity paradox of information technology. *Communications of The ACM* **36(12)**, 67-77.

Carmel E (1997) American hegemony in packaged software trade and the "culture of software. *The Information Society* **13(1)**, 125-142.

Carmel E (1995) Cycle-time in packaged software firms. *Journal of Product Innovation Management* **12(2)**, 110-123.

Carmel E and Becker S (1995) A process model for packaged software development. *IEEE Transactions on Engineering Management,* **41(5)**, 50-61

Carmel E and Bird B (1997) Small is beautiful: a study of packaged software development teams. *Journal of High Technology Management Research,* **8(1)**, 129-148.

Carmel E and Sawyer S (1998) Packaged software development teams: What makes them different? *Information Technology & People,* **11(1)**, 7-19.

Crowston, K., Annabi, H., & Howison, J. (2003). *Defining Open Source Software Project Success.* Paper presented at the International Conference for Information Systems, Seattle.

Crowston, K., Annabi, H., Howison, J., & Masango, C. (2004). *Innovative systemic perspectives: Effective work practices for software engineering: free/libre open source software development.* Paper presented at the 2004 ACM workshop on Interdisciplinary software engineering research.

Cubranic, D., & Booth, K. S. (1999). *Coordinating Open-Source Software Development.* Paper presented at the Proceedings of the 7th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises

Cusumano M and Smith S (1997) Beyond the waterfall: Software development at Microsoft," in *Competing in the Age of Digital Convergence* (Yoffie D, Ed.), 371-411. Boston: Harvard Business School Press.

Drucker, P. F. (1998). Management's New Paradigms. *Forbes, 162*, 152-177

Egyedi, T. M. (2004). Standardization and other coordination mechanisms in open source software. *International Journal of IT Standards & Standardization Research, 2*(2), 1-17.

Fielding, R. T. (1997). *The Apache Group: A case study of Internet collaboration and virtual communities*. Retrieved April 26, 2005, from http://roy.gbiv.com/talks/ssapache/title.htm

Fielding, R. T. (1999). Shared Leadership in the Apache Project. *Communications of the ACM, 42*(4), 42-43

Goodman, P., E. Ravlin, and L. Argote. (1986) "Current Thinking About Groups: Setting the Stage for New Ideas", in P. Goodman and Associates (eds.), *Designing Effective Work Groups*, Jossey-Bass, San Fransisco, .

Grudin J (1991) Interactive systems: Bridging the gap between developers and users. *IEEE Computer* **24 (5)**, 59-69.

Halloran, T. & Scherlis, W, (2002) High Quality and Open Source Software Practices, *Proc. 2nd Workshop on Open Source Software Engineering*, May, Orlando, FL,

Hecker, F. (1999). *Mozilla at one: A look back and ahead*, from http://www.mozilla.org/mozilla-at-one.html

Humphrey, W. (1989). *Managing the software process*. Reading, MA: Addison-Wesley.

Keil M and Carmel E (1995) Customer-developer links in software development. *Communications of the ACM* **38(5)**, 33-44.

Kling, R. (1999) What is social informatics and why does it matter? *D-Lib Magazine*, 5(1). Retrieved from http://www.dlib.org:80/dlib/january99/kling/01kling.html

Kling, R. (2000). Learning about Information Technologies and Social Change: The Contribution of Social Informatics. *The Information Society*, *16*(3), 212-234.

Kling, R., Rosenbaum, H. and Sawyer, S. (2005), *Understanding and Communicating Social Informatics: A Framework for Studying and Teaching the Human Contexts of Information and Communication Technologies*. Medford, NJ: Information Today.

Kling, R., & Iacono, S. (1984). The control of information systems developments after implementation. *Communications of the ACM, 27*(12), 1218-1226.

Law, J. & Bijker, W. (1992). Technology, Stability and Social Theory. In W. Bijker (Ed.) *Shaping Technology/Building Society*. Cambridge, MA: MIT Press: 32-50.

Maiden, N. and Ncube C (1998) Acquiring COTS software selection requirements. *IEEE Software* **15(2)**, 46-56.

Markus, M. (1983). Power, Politics, and MIS Implementation. *Communications of the ACM, 26*(6), 430-444.

Melone, N. (1990). A theoretical assessment of the user-satisfaction construct in information systems research. *Management Science, 36*(1), 76-91.

Merton, R. (1967). *On theoretical sociology*. New York: The Free Press.

Mi & Scacchi, W, (1993) Articulation: An Integrated Approach to the Diagnosis, Replanning, and Rescheduling of Software Process Failures , *Proc. 8th. Knowledge-Based Software Engineering Conference*, Chicago, IL, IEEE Computer Society, 77-85.

Mockus, A., Fielding, R. T., & Herbsleb, J. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology, 11*(3), 309-346.

Mockus, A. & Herbsleb, J. (2002) Why not improve coordination in distributed software development by stealing good ideas from Open Source? , *Proc. 2nd Workshop on Open Source Software Engineering*, May, Orlando, FL.

Moody, G. (2001). *Rebel code—Inside Linux and the open source movement*. Cambridge, MA: Perseus Publishing

Moon, J. Y., & Sproull, L. (2000). Essence of distributed work: The case of Linux kernel. *First Monday, 5*(11) Available online at …

Raymond, E. S. (1998). The cathedral and the bazaar. *First Monday, 3*(3)

Sawyer, S., (2004) "Software Development Teams: Three Archetypes and their Differences." *Communications of the ACM, 17*(12), 92-97.

Sawyer, S. (2001) "Information Systems Development: A Market-Oriented Perspective," *Communications of the ACM, 44*(11), 97-102.

Sawyer, S. (2000) "Packaged Software: Implications of the Differences from Custom Approach-es to Software Development," *European Journal of Information Systems, 9*(1), 47-58.

Scacchi, W. (2005) Socio-Technical Interaction Networks in Free/Open Source Software Development Processes, in S.T. Acuña and N. Juristo (eds.), *Software Process Modeling*, Springer Science+Business Media Inc., New York, 1-27.

Scacchi, W. (2002). Understanding the requirements for developing open source software systems. *IEE Proceedings-Software, 149*(1), 24-39.

Scacchi, W. (2002b) Process Models in Software Engineering , in J. Marciniak (ed.), *Encyclopedia of Software Engineering* (2cd edition), Wiley, New York, 993-1005

Shaikh, M. and Cornford, T. (2004)Version Control Tools: A Collaborative Vehicle for Learning in F/OS, *Proceedings of the 26th International Conference on Software Engineering proceedings - Collaboration, Conflict and Control: The 4th Workshop on Open Source Software Engineering*, May 25, Edinburgh, Scotland..

Torvalds, L. (1999). The Linux edge. *Communications of the ACM, 42*(4), 38-39

Vessey, I., & Sravanapudi, P. (1995). Case tools as collaborative support technologies. *Communications of the ACM, 38*(1), 83-95.

Weick, K. (1995). What theory is not: Theorizing is. *Administrative Science Quarterly, 40*, 385-390.