Modeling Dynamics in Agile Software Development

LAN CAO
Old Dominion University
BALASUBRAMANIAM RAMESH
Georgia State University
and
TAREK ABDEL-HAMID
Naval Postgraduate School

Changes in the business environment such as turbulent market forces, rapidly evolving system requirements, and advances in technology demand agility in the development of software systems. Though agile approaches have received wide attention, empirical research that evaluates their effectiveness and appropriateness is scarce. Further, research to-date has investigated individual practices in isolation rather than as an integrated system. Addressing these concerns, we develop a system dynamics simulation model that considers the complex interdependencies among the variety of practices used in agile development. The model is developed on the basis of an extensive review of the literature as well as quantitative and qualitative data collected from real projects in nine organizations. We present the structure of the model focusing on essential agile practices. The validity of the model is established based on extensive structural and behavioral validation tests. Insights gained from experimentation with the model answer important questions faced by development teams in implementing two unique practices used in agile development. The results suggest that due to refactoring, the cost of implementing changes to a system varies cyclically and increases during later phases of development. Delays in refactoring also increase costs and decrease development productivity. Also, the simulation shows that pair programming helps complete more tasks and at a lower cost. The systems dynamics model developed in this research can be used as a tool by IS organizations to understand and analyze the impacts of various agile development practices and project management strategies.

Authors' addresses: L. Cao, College of Business, Old Dominion University, 5115 Hampton Boulevard, Norfolk, VA 23529; B. Ramesh (corresponding author), J. Mack Robinson College of Business, Georgia State University, P.O. Box 3965, Atlanta, GA 30302-3965; email: bramesh@gsu.edu; T. Abdel-Hamid, Information Sciences Department, Naval Postgraduate School, 1 University Circle, Monterey, CA 9343.

DOI 10.1145/1877725.1877730 http://doi.acm.org/10.1145/1877725.1877730

Categories and Subject Descriptors: I.6.0 [**Simulation and MOdeling**]: General

General Terms: Design, Experimentation, Economics

Additional Key Words and Phrases: Agile software development, simulation, process modeling, system dynamics

ACM Reference Format:
Cao, L., Ramesh, B., and Abdel-Hamid, T. 2010. Modeling dynamics in agile software development. ACM Trans. Manag. Inform. Syst. 1, 1, Article 5 (December 2010), 26 pages.
DOI = 10.1145/1877725.1877730 http://doi.acm.org/10.1145/1877725.1877730

1. INTRODUCTION

Changes in the business environment such as turbulent market forces, rapidly evolving system requirements, and advances in technology demand agility in the development of software systems. In response, many firms have replaced their traditional plan-driven development with more adaptive, agile approaches. According to a recent survey by Forrester Research [West et al. 2010], agile processes have joined the mainstream of information systems development approaches with nearly half of the IT professionals using some agile development practices. Though agile approaches have received wide attention, empirical research that evaluates their effectiveness and appropriateness is scarce. Much of the research to-date has focused on studying the impacts of select agile practices such as pair programming [Williams and Kessler 2000] and test-driven development [Erdogmus and Williams 2003]. However, we argue that the dynamic nature of agile practices requires that they be studied as an integrated system rather than as individual (isolated) practices. For example, agile practices magnify the impacts of dynamic project features such as feedback, (e.g., iterative feedback from customers), time delays (e.g., delays in implementing change requests), and nonlinear cause-effect relationships among project components (e.g., relationship between schedule pressure and adding new personnel). Successful management of agile projects requires an understanding and exploitation of such dynamic features. Extant literature does not help integrate the understanding of individual practices and their implications for the entire development process. Motivated by this concern, our research uses system dynamic simulation to investigate the impacts of agile practices by modeling them as a dynamic system.

Although there is strong interest among both researchers and practitioners on the use of agile methods, current research is fragmented and is restricted to the study of specific practices. Beyond some case studies and surveys (for example, Cao et al. [2009], Fitzgerald et al. [2006], Grenning [2001], Ramesh et al. [2010], Rumpe and Schroeder [2002]), the effectiveness and applicability of agile methods have not been adequately investigated [Glass 2004]. For example, Boehm [2002] observes that it is difficult to scale up agile methods for large projects because agile development often lacks sufficient architecture planning, intensely focuses on early results, and may have low test coverage. He also recommends against using agile methods in the development of mission-critical

systems. The applicability of agile approach is constrained by several factors such as project size and type, experience of project personnel, and the availability of knowledgeable and committed customers [Erickson et al. 2005; Fitzgerald et al. 2006]. These studies highlight some fundamental questions that have not yet been adequately investigated. These include the following: How does the cost of incorporating changes in the system with agile development compare with traditional approaches which typically experience exponentially increasing costs over the system development lifecycle? Are critical agile practices such as pair programming cost effective?

To address these (and similar) questions, we study agile software development by modeling it using System Dynamics (SD) simulation. SD modeling of agile development will help answer difficult but critical questions that researchers and managers face in balancing the needs for quality, cost effectiveness, and speed in software development. The SD model developed in our research focuses on essential aspects of agile software development. Our model is developed on the basis of an extensive review of the literature, focused field interviews in nine organizations that use agile methods, and secondary data.

In the next section, we present the model structure. The following six major components that represent the essential aspects of agile development are described: customer involvement, change management, agile planning and control, refactoring and quality of design, software production, and human resource management. Then we describe an extensive validation of the model, using both structural and behavioral validation tests. We also present insights gained from experimentation with the model to answer critical questions faced by IS managers and developers, specifically investigating two important agile development practices: refactoring and pair-programming policies. Finally, we present the conclusions.

2. SYSTEM DYNAMICS MODELING OF SOFTWARE DEVELOPMENT

Simulation is increasingly used to understand software development processes because it provides a viable laboratory tool to study the complex phenomenon that is hard to investigate with other tools. Simulation has been used to investigate a variety of aspects in IS development such as reliability [Rus et al. 1999], requirements management [Höst et al. 2001], and testing [Cangussu 2004]. System Dynamics (SD), first developed by Jay Forrester [Forrester 1961], is a modeling methodology that uses feedback control systems principles and techniques to represent the dynamic behavior of the managerial, organizational, and socio-economic systems. SD has been used widely in many research areas including urban planning, public policy, economics, management, decision making, healthcare, and organizational learning.

System dynamics uses feedback controls to reflect the interconnection between elements in a system. Feedback systems can capture the interconnection between agile practices in a closed sequence of causes and effects. The simulation model provides an integrated environment within which the impact of an agile practice on the entire development process can be evaluated, mimicking the real-life environment. Agile practices are tightly related to each other. For

example, refactoring is supported by unit testing, continuous integration, and simple design, while unit testing is supported by pair programming. The dynamic nature of agile practices requires that they be studied as an integrated feedback system rather than as individual (isolated) practices. SD provides an environment in which such interconnection between agile practices can be fully represented and studied.

The use of SD methodology in traditional software project management [Abdel-Hamid and Madnick 1991] represents the first comprehensive use of this method in IS research. This research develops an extensive and integrative model that covers issues in human resource management, software production, and planning and control. This model integrates multiple functions of the software development process, and includes both management-type functions (e.g., planning, control, and staffing) and software production-type activities (e.g., design, coding, review, and testing). This model has been used to investigate a wide range of areas in software development including software cost and schedule estimation [Abdel-Hamid 1990, 1993a; Abdel-Hamid and Madnick 1983], the economics of the quality assurance function [Abdel-Hamid and Leidy 1991], project staffing [Abdel-Hamid 1989; Abdel-Hamid et al. 1994; Sengupta et al. 1999], software reuse [Abdel-Hamid 1993b] and project control with fallible information [Abdel-Hamid et al. 1993; Sengupta and Abdel-Hamid 1996]. More recently, SD modeling has been used extensively in research on the software development process [Madachy 2007] including requirements engineering [Stallinger and Grünbacher 2001], reliably control [Rus et al. 1999], outsourcing [Dutta and Roy 2005; Roehling et al. 2000], information security [Dutta and Roy 2008], knowledge management [Peters and Jarke 1996], and system acquisition activities [Choi and Scacchi 2001]. To the best of our knowledge, our work is the first to apply SD simulation to model essential aspects of agile software development.

3. MODEL STRUCTURE

3.1 Data Sources

Primary data collected from agile development projects was used to develop the model structure and specify its parameters. Data from nine organizations was used to build and refine the model. These organizations use eXtreme Programming (XP) or SCRUM, or a combination of the two methods. In addition, we also used secondary data sources (such as project documentation as well as online news groups devoted to agile development) which provided access to extensive discussions on agile practices and principles. Tables I and II provide details on organizations studied and the secondary data sources used. Each is identified with a letter code. The description of our model refers to relevant data sources used in its development.

The SD model developed in this research integrates essential practices in agile development such as agile planning, short iterations, customer involvement, refactoring, unit testing, and pair programming. These practices are modeled in four submodels/sectors (Figure 1): customer involvement, change management,

Table I. Profile of Interviewees and Organizations

Phase I: Initial Model Development			
Organization*	Industry and Product	Project Information	Participants
NetworkSoft (N)	Network Software Consulting. Offers services on developing network systems, and architectures. 150+ IT personal.	Project duration: 8 month Team size: average 12 people Method: XP	Senior Software Engineers (N1, N2, N3) Project Manager (N4)
SecuritySoft (S)	Security Software. Offers software for Internet security. Acquired by a large IT company which has more than 350k employees in 2005.	Project duration: 6 month Team size: 10 people Method: XP	Product Manager (S1) Project Manger (S2) Senior Software Engineers (S3, S4, S5) QA (S6)
DataSoft (D)	Across several industries. Offers data management online. 30+ employees.	Project duration: 7 month Team size: 5 people Method: XP	Technical Lead (D1)
EbizSoft (E)	Packaged Software Development. Offers e-Business connections and transactions. 3000 product specialists globally.	Project duration: 10 month Team size: 15 people Method: XP and SCRUM	Project Manager (E1) Developers (E2, E3, D4)
HealthSoft (H)	Healthcare Information Systems. Offers software to healthcare facilities. 200+ IT professionals.	Project duration: 8 month Team size: 10 people Method: XP	Senior Software Engineers (H1, H2, H3)
Phase II: Model Refinement			
ConsultSoft (C)	Software Consulting. Offers consulting services on software development. 1200+ employees globally, leading agile development company.	Project duration: 12 month Team size: 50+ people Method: XP	Team Lead (C1) Project Manager (C2)
BankSoft (B)	Banking Information Systems. Offers software that handles financial transactions. 50+ employees	Project duration: 36 month Team size: 22 people Method: XP	Developers (B1, B2, B3, B4) Technical Lead (B5)
WebSoft (W)	Across Industries. Offers to help Brick & Mortar companies develop web presence. 20+ employees	Project duration: 5 month Team size: 4 people Method: XP	General Manager (W1)
FinSoft (F)	Online Financial Transactions support. Offers online payments. 50+ employees	Project duration: 8 month Team size: 6-9 people Method: XP and SCRUM	Project Manager (F1) Developers (F2, F3, F4)

*Pseudonyms are used to protect the identity of the participating organizations and interviewees.

Table II. Secondary Data Sources

Type	Description
Project documentation (PD)	Design documents, estimation and actual effort data, defect metrics, status metrics
Agile user group meetings (UM)	Monthly meetings to discuss issues in agile software development. The authors attended the meetings for over three years.
Online news group on agile methods (NA1)	http://groups.yahoo.com/group/xp-location (redacted).
Online news group on agile methods (NA2)	http://groups.yahoo.com/group/extremeprogramming
Direct correspondence (DC)	Emails, phone calls and instant massages with agile developers (other than those listed in table I)

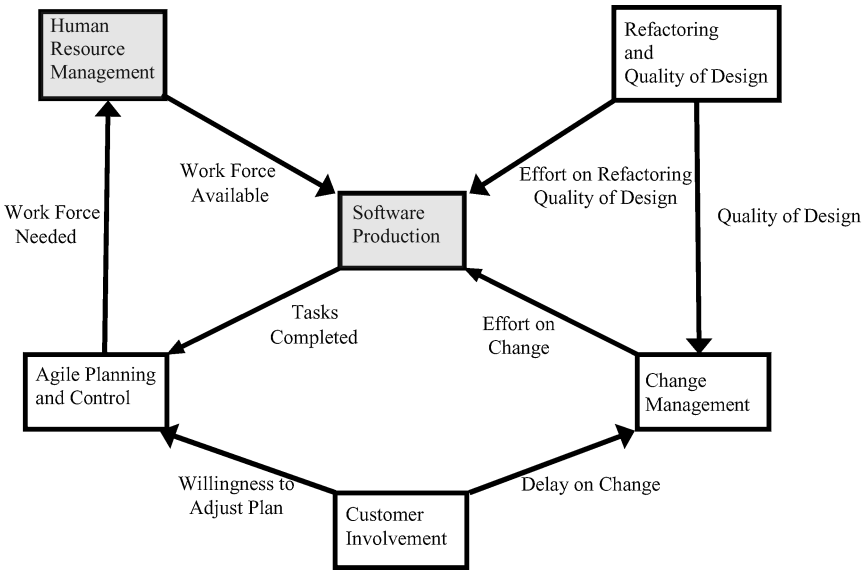
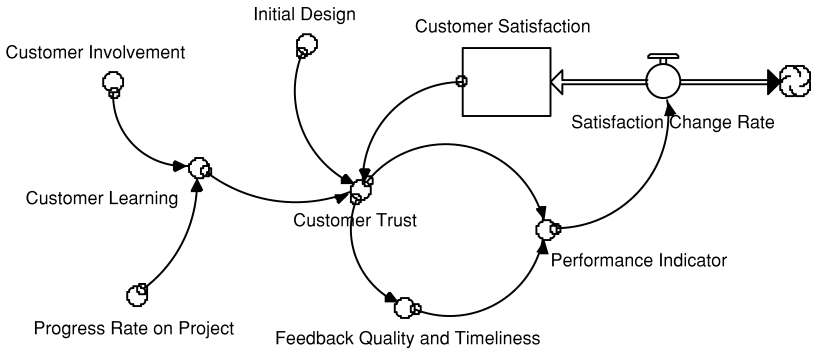


Fig. 1. Model structure.

agile planning and control, and refactoring and quality of design. Human resource management and software production subsystems that are necessary to support agile development (shown in grey) were created by adopting and extending the work of Abdel-Hamid and Madnick [1991]. Due to space constraints, we provide brief descriptions of all the subsystems, and elaborate only on refactoring and quality of design. A major extension of the human resource subsystem system in our work involves the modeling of pair programming, which is unique to agile development. It is described in the online appendix.

3.2 Customer Involvement

Agile development relies on direct face-to-face communication between customers and developers for knowledge sharing [Martin 2000]. For example, XP insists on having an on-site customer [Beck 2000] whose responsibilities include understanding and representing the needs of multiple customer segments, specifying and clarifying the features that need to be implemented,



Legend:

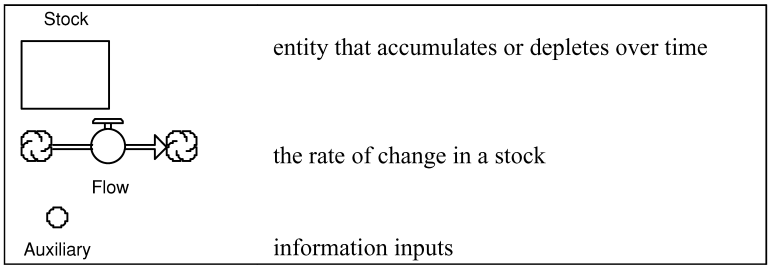


Fig. 2. Model structure for customer involvement.

developing acceptance tests and verifying that these tests are run correctly, and participating in the planning of iterations and releases. The success of an XP project highly depends on the ability of the on-site customer to adequately fulfill these responsibilities. Boehm and Turner [2004] identify several characteristics of a "good" customer: Collaborative, Representative, Authorized, Committed, and Knowledgeable (CRACK). Agile methods such as XP thus impose a heavy burden on the customer and assume the availability of such capable customers. The use of an on-site customer poses several challenges, including the lack of customer competency, degree of involvement, and conflicts of interest among different customers.

Figure 2 shows the customer involvement subsystem. Customer learning about the important aspects of the project and the development process is affected by the level of customer involvement and rate of progress of the project [Curtis et al. 1988]. Customer learning, in turn, leads to customer trust. The higher the customer trust, the higher the quality and speed of the feedback provided by the customer to the development team. Delayed feedback, on the other hand, may increase the cost of the project because changes that are made late in the lifecycle are more difficult to implement. The increased cost is reflected in the perceived performance of the team (modeled as the performance indicator). The decline in performance and/or inadequately implemented changes results in decreased customer satisfaction, which in turn affects customer trust. In addition, the lack of detailed initial design has a negative impact on building initial customer trust.

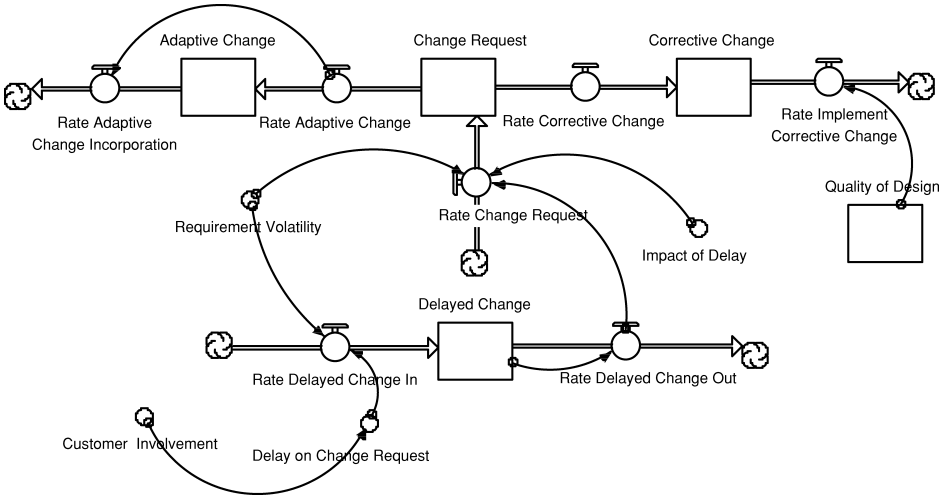


Fig. 3. Change management.

Customer trust is essential in agile development because the development team relies on the customer to make important decisions on many issues such as prioritization of product features and project scope or schedule adjustment. Unwillingness of the customer to adjust the scope of the project will increase the schedule pressure on the developers, which has been known to cause poor-quality designs and code.

3.3 Change Management

This subsystem models how agile development handles changes caused by requirements volatility. Embracing change is a core characteristic of agile methods. The alignment of the development process with a changing environment is a critical motivation for adopting agile methods. As new requirements are added to a release, or as existing requirements are deleted or modified, the product's cost, schedule, and quality are impacted. Figure 3 shows the change management process in agile development.

The volume of changes is decided by requirements volatility, defined as the ratio of changed requirements (added, deleted, and modified) to total requirements [Stark et al. 1999]. Changes fall naturally into three main classes [Basili and Weiss 1984; Lientz et al. 1978; Swanson 1976]: adaptive, corrective, and perfective. In agile development, perfective change is typically implemented through refactoring. Therefore, only adaptive and corrective changes are modeled in this subsystem. Adaptive change requests are treated as new requirements that are scheduled for future iterations, unless the requested change significantly affects the scope, schedule, and cost constraints set by the customer or management [Highsmith and Cockburn 2001].

Corrective change requests are implemented in the order of their assigned priority. The rate at which corrective changes are implemented varies depending on the quality of the design (as described in the subsystem: refactoring

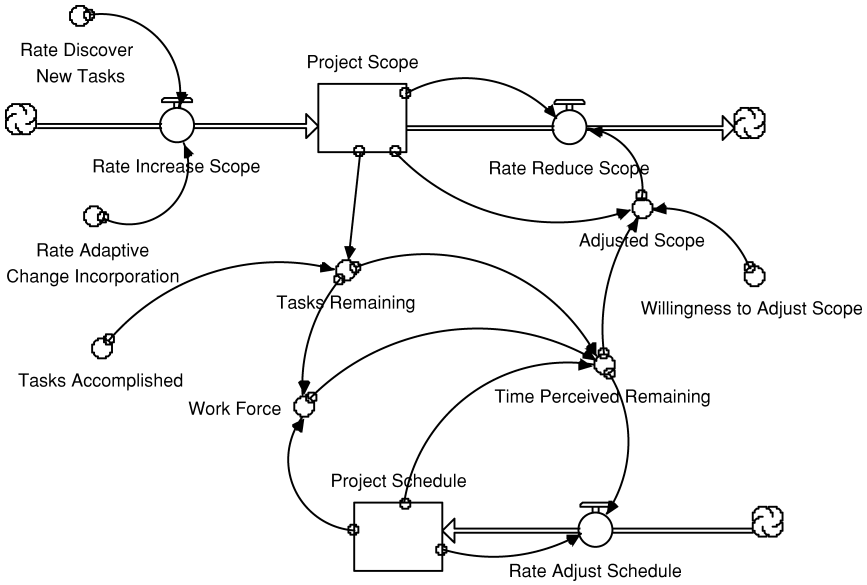


Fig. 4. Planning and control.

and quality of design). The customer reviews the features as soon as they are implemented and suggests changes that need to be incorporated in the next iteration [Beck 2000; Highsmith and Cockburn 2001]. However, the timeliness with which changes are identified depends on customer involvement. A more involved customer will request changes earlier, while a less involved customer may only realize the need for changes after some delay. Both the literature [Elssamadisy and Schalloil 2002] and our interviews confirmed the significant effect of delayed customer feedback (N3, N4, D1, F1, F2). Therefore, at any stage of the project, corrective changes requested by the customer include changes to the features that have been just developed and those developed in previous iterations. Delayed changes require more effort to implement because of their ripple effects which are modeled by the variable "Impact of Delay".

3.4 Agile Planning and Control

The planning and control subsystem (Figure 4) adjusts the schedule and scope of the project based on the progress of the project. It includes two critical processes: scope adjustment and schedule adjustment.

Scope adjustment. In contrast to traditional development, in agile development projects, scope is an extra lever that can be readily manipulated [Beck 2000]. As customers provide feedback continuously and request new features, project scope is continuously adjusted. Previously planned features can be removed or deferred when the project is behind schedule. In agile planning, project scope is adjusted as follows (especially when working under schedule pressure).

(1) System functionalities are categorized into two groups: the must-have or the most important features (stories) which must be implemented first and the nice-to-have features which may be implemented later. Under schedule pressure, nice-to-have features are often dropped.

(2) Stories are decomposed into finer pieces, called tasks. Only critical tasks are implemented and the others are dropped when the project is under schedule pressure.

When the team realizes that the project scope needs to be adjusted to meet the delivery schedule, it negotiates with the customer to identify features that may be eliminated. The customers' "willingness to adjust scope" depends on "customer trust" which is described in the customer involvement subsystem.

Schedule adjustment. In agile software development, schedule is adjusted based on the following constraints.

(1) If a project has a fixed completion date and fixed resources, then the schedule is fixed as initially planned. If the project is behind schedule, the scope needs to be reduced by dropping stories or tasks, since it is usually difficult to change the workforce (and indirectly the budget).

(2) Similar to a traditional development project [Abdel-Hamid and Madnick 1991], an agile project will first adjust the schedule and then the cost. Then, the agile development team often has the flexibility to negotiate changes in the scope of the project with the customer by canceling or deferring some tasks from the current release.

3.5 Refactoring and Quality of Design

This subsystem (shown in Figure 5) models refactoring and its impacts on the quality of design. In highly volatile environments, detailed up-front design is considered wasteful because much of the initially identified functionality may never be fully implemented due to changes in requirements. Refactoring is used to keep the quality of design at just the acceptable level that meets current needs. This practice, often referred to as restructuring [Arnold 1986], has been used to reduce software complexity by incrementally improving internal software quality. In agile development, the intensity with which refactoring is practiced, especially due to the lack of detailed initial design, makes it very critical and unique. Fowler [1999] defines refactoring as "[a] change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." In object-oriented development, for example, the key idea behind refactoring is to redistribute classes, variables, and methods across the class hierarchy in order to facilitate future adaptations and extensions [Mens and Tourwe 2004; Opdyke 1992]. Using refactoring, a software developer can improve the design of software, make software easier to understand, find bugs, and develop programs faster [Fowler 1999]. Agile methods rely on refactoring to build up the design continuously during development, instead of spending much effort on up-front design. Fowler [1999] claims that refactoring "leads to a program with a design that stays good as development continues." Continuous and incremental refactoring is a mechanism used to incorporate frequent changes.

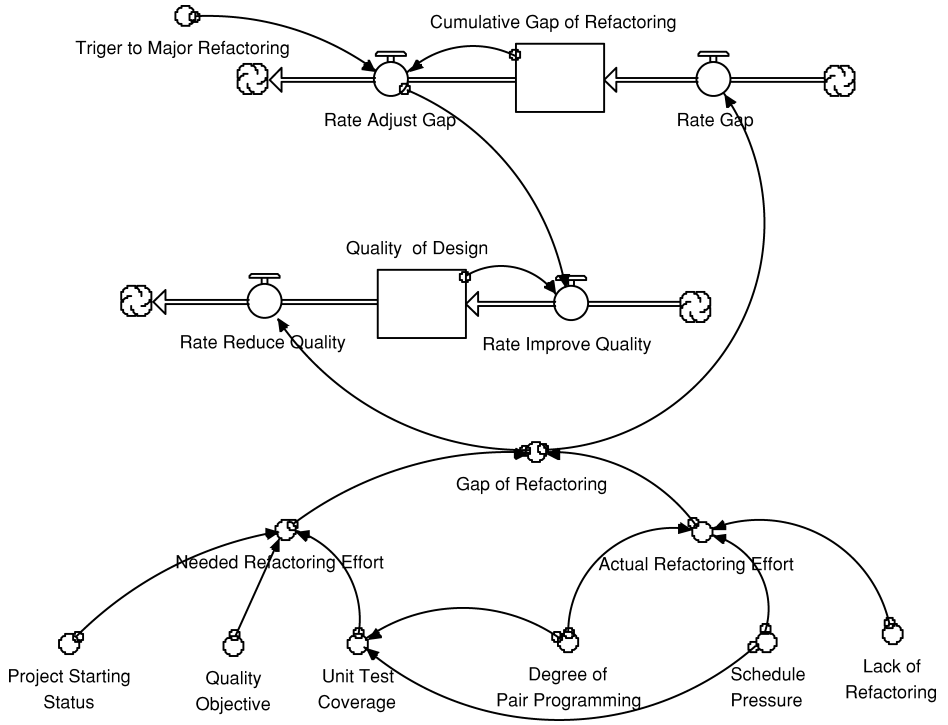


Fig. 5. Refactoring and quality of design.

Needed vs. Actual Refactoring Effort.

Needed refactoring effort. Refactoring consumes a major portion of the development effort if it is done regularly. Empirical studies on refactoring [Alshayeb and Li 2005; Li and Henry 1993; Stroulia and Leitch 2003] estimate that the actual effort ranges from 6% to 39% of the total development effort. Also, these studies note that many programmers mix refactoring and new work and therefore, it is difficult to measure the actual amount of refactoring effort. Our data (NA1, NA2, DC) also suggests that it is difficult for developers to separate refactoring from other development. Before implementing a new feature, often developers refactor existing code. In the model, we use the average effort reported by the study participants, which at 30% represents a significant part of the total development effort.

Three factors affect the effort needed for refactoring: quality objectives, the starting status of the project [Harrison 2003; Hodgetts 2004], and the coverage of unit tests [Beck 2000; Beck 2003; George and Williams 2002; Jeffries et al. 2001]. Quality objectives refer to the expected quality goals of the project. The starting status refers to the quality of the up-front design before coding starts. Unit tests provide a safety net of regression tests and validation tests so that refactoring and integration can be done effectively. Without sufficient coverage of unit tests, more effort is needed to validate the code which may change due to refactoring. Unit tests are used to test the correctness of a particular module

of source code [Beck 1994]. A common practice is to write test cases for every nontrivial function or method in the module. Unit tests are written before or in conjunction with production code; they are maintained with production code and are used by every developer working on the software.

The effort needed to make a single change is much higher when unit testing is not done. One developer (C1) described his experience of a project without unit testing as follows: ". . . any change in one area, we had to test three or four use cases because we were reusing the code as much as we could. So you make one change, you have to walk through three or four use cases. And that slowed us down because we didn't have the unit tests." The relationship between test coverage and the developer's confidence in the software and the effort needed to make changes is not linear. One developer (N2) describes his experience on a project that has partially covered unit testing as follows: "the benefit of unit testing is not proportional to unit test coverage. If unit test coverage is low, then the benefits are significantly low." The actual coverage of unit tests depends on the planned coverage as well as the degree of schedule pressure and pair programming. Under schedule pressure, developers will reduce unit testing and focus more on coding [Beck and Gamma 1998]. Paired developers are generally more disciplined than individuals in writing unit tests [Williams 2001].

Actual refactoring effort. Developers almost never spend enough effort on refactoring [Opdyke 1995]. In the model, insufficient refactoring is modeled using the variable "lack of refactoring." Our interviews (S3, S5, H1, B5) with agile developers confirmed this tendency. Our data (S3, S5, H1, B5, PD, UM) also highlighted the effects of schedule pressure and pair programming on refactoring. The effect of schedule pressure on refactoring effort has been observed in several studies [Berry 2002; Fowler 1999; Opdyke 1992; Whiler 2003]. In our interviews (N4, E1), project managers acknowledged that refactoring will be largely ignored when working under a tight schedule. Also, prior studies [Beck 2000; Williams and Kessler 2000] observe that pair pressure can force developers to frequently refactor the code. Our data (S1, S2, S6, H2) suggests that developers will decrease their refactoring activity by about 50% if not paired with others. When the project is close to the deadline, developers choose not to refactor because productivity gain from refactoring would be realized only after the deadline [Fowler 1999].

The gap between the needed and actual refactoring effort, which may be considered a "technical debt" [Cunningham 1992], leads to unclean code, which in turn leads to corrupt designs. As requirements change during development, the design is not updated to encompass the new requirements since changing the design would involve refactoring large amounts of code. Therefore, changes are made in the most expedient manner possible and the design of the system becomes increasingly corrupt and brittle. Then, the developers have to perform "major refactoring," that is investing a large amount of effort to clean up the code. The deferred refactoring requires extra effort to clean up the code later [Cunningham 1992; Elssamadisy and Schalloil 2002] because of the large number of features involved [Gorts 2004].

Quality of design. In agile development, it is claimed that the "source-code is the design," design activity is part of the programming process, and design

evolves with the code [Shore 2004]. The effect of refactoring on maintainability has been evaluated using metrics such as coupling [Kataoka et al. 2002]. However, in agile development, Cockburn points out "there is no agreed upon measure of quality of a design. Therefore there is no single-valued dimension to 'improve' along. The quality of a design is subject (not only to the viewer's preconceptions) to the circumstances in which it is located—its purpose and future life."[1] For this reason, in the model, we represent the perception of the quality of design as a variable with values ranging from 0 to 1.

The quality of design is modeled as a stock. The initial value is set to 1, which means that at the beginning of the project, the quality of design is at its desired value since no activities that deteriorate the design have been conducted. As the project progresses, this high level of quality needs to be maintained by refactoring. Insufficient refactoring reduces the quality of design while major refactoring brings the quality of design back to an acceptable level.

3.6 Software Production and Human Resource Management

Human resource management and software production subsystems are adapted from the work of Abdel-Hamid and Madnick [1991]. A major extension is the modeling of pair programming, which is unique to agile development. Nosek [1998] examined the role of collaboration in a study of 15 full-time, experienced programmers working on a challenging problem. This study concludes that collaboration improves both the programmers' performance and their enjoyment of the problem solving process. Williams et al. [2000] conducted a structured experiment comparing pair programming with solo programming in a classroom setting. The results show that pair programming yields a better product with higher quality in less time and that the developers using this approach enjoy the process more. However, pair programming allocates two persons to each task, which increases labor costs. Therefore, its cost effectiveness over the project lifecycle requires examination.

—*Pair Programming and Productivity.* Although pair programming uses two persons in each task, some studies find that paired developers take less time on each task. The results from empirical studies on the overall impact of pair programming on productivity are mixed [Karlström 2002; Parrish et al. 2004; Williams and Kessler 2000; Williams et al. 2000]. Therefore, we used the data from our interviews (N1, N3, E2, C1) which suggest that pair programming in a typical project consumes about 30% more effort than solo programming.

—*Pair Programming and Rework.* Prior studies report that pair programming results in 40% to 90% fewer defects [Erdogmus and Williams 2003; Williams et al. 2000]. Our data (S1, S2, S4, E1) collected from real projects indicates that pair programming results in the reduction of defects by more than 50%.

—*Pair Programming and Refactoring and Unit Testing.* As discussed earlier, pair programming also impacts the effort spent on refactoring and unit testing. Our interviews (S1, H3) indicate that when developers are working alone, the probability that they will engage in refactoring is reduced by half

[1](http:/c2.com/cgi/wiki?WhyDidYouRefactorThat).

because they are not likely to be as disciplined as a paired team. Similarly, paired developers are more disciplined than individuals in writing unit tests. Our data (N4, S3) suggests that when developers are working individually, they will reduce the coverage of unit testing.

In summary, our SD model represents essential practices used in agile development. It represents agile development as an integrated system of practices rather than in isolation, and incorporates dynamic features such a feedback, time delays, and nonlinear cause-effect relationships. Therefore, this model can be used to examine the dynamic nature of the impact of various practices on critical project outcomes such as cost, schedule, and quality.

3.7 Model Validation

Our model has been thoroughly validated after it was developed and refined. Model validation is concerned with creating sufficient confidence in a model for its results to be acceptable. System dynamics research emphasizes a wide range of tests, including tests of model structure, and the ability of the model to reproduce real-life behavior [Barlas 1989; Forrester 1961; Forrester and Senge 1980; Sterman 2000]. The identification of the appropriate structure is the first step in establishing validity of a SD model because the structure drives its behavior [Forrester 1961]. Once the structural validity is sufficiently established, behavior validity, that is, how well the model-generated behavior mimics the observed behavior of the real system, is assessed to achieve the overall validity of the model [Sterman 2000].

In the online appendix, we report in detail the procedures and results of our extensive model validation. First, results from structural validation are presented. In his seminal work Sterman [2000] identifies a dozen tests that may be used for structural validation. A variety of tests that focus on different aspects of a model are used to ensure that the model satisfies the purpose for which it was developed [Sterman 2000]. It should be noted that there is no single test that can be used to either validate or invalidate the structure of a model. The following nine tests that were relevant for the study were used in our validation: boundary adequacy, structure assessment, dimensional consistency, parameter assessment, extreme conditions, behavior anomaly, and surprise behavior. The results from all the structural validation tests suggest that our model structure closely reflects real agile development processes. Then, we conducted behavioral validation using data from an independent case study of a real-life agile development project. The objective of the case study was to examine the model's ability to reproduce the dynamic patterns observed in an agile software development project.

The behavioral validation was done with data from RealSoft[2] located in a Rocky Mountain state. This company is not only a major vendor of tools used in agile development, but it also uses these tools and agile methods in its development process. The project followed an iterative approach, with each iteration lasting two weeks. At the time of data collection, the project was

[2]To ensue anonymity of the participating organization, a pseudonym is used.

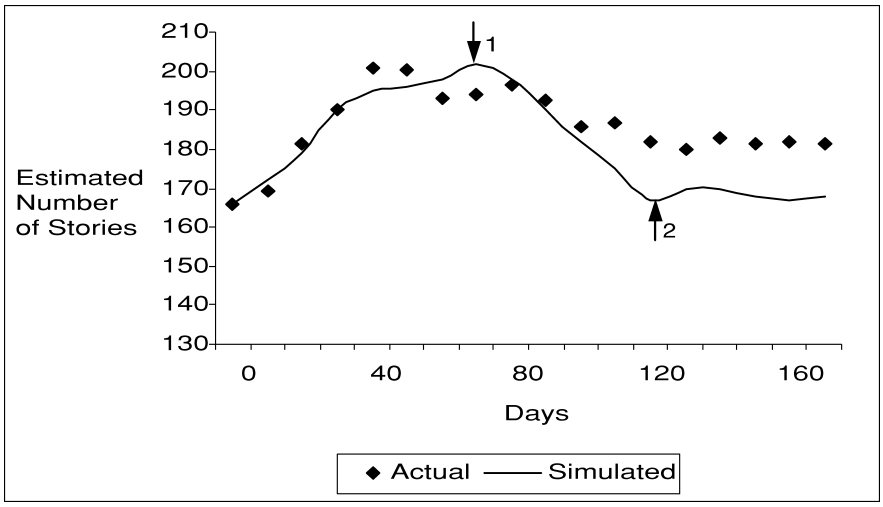


Fig. 6. Estimated number of stories (simulated and actual) of RealSoft project.

ongoing and had plans for several more releases in the future. The development teams consisted of four developers, one quality assurance engineer, and one product manager (who also acted as a surrogate customer). Other relevant details on the organization and project are described in the online appendix.

The parameters of the model were set to values that reflected the RealSoft environment. Once the model was parameterized, it was run to simulate the RealSoft project in the STELLA® simulation environment. We examined the dynamic behavior of the following three critical project variables: (1) estimated number of stories, (2) productivity, and (3) number of defects. We compare the model's simulated behavior over time with actual project results. The model replicates the actual RealSoft project behaviors closely in the three variables examined. Due to space limitation, we only present results on the examination of the first variable. The remaining results are presented in the online appendix.

Estimated number of stories. Figure 6 depicts the actual and simulated estimated number of stories. The figure shows that the estimated project scope changed over time. The growth and decline of scope are caused by two factors which impact scope adjustment in opposite directions: (1) scope extension: the scope tends to increase as new tasks are discovered and the customer requests new features, (2) scope shrinkage: the scope is reduced to meet the deadline because the project is considered to be behind schedule. The scope is extended or reduced depending on the factor that has a stronger impact. In this case study, scope extension is dominant during the first 80 days (marked by arrow 1) because the completion date is extended during that period. However, when the maximum tolerant completion date is reached, the impact of the schedule constraint is greater and the scope is reduced to meet the scheduled delivery date.

The model estimates the number of stories to be 168, which is slightly lower than the actual for the last two releases (181). A plausible reason for this slight difference is that the model assumes that the project concludes at the end of the modeled cycles. Towards the end of the project, management is

more likely to adjust the scope of the project to accommodate deadlines. In the model, the customer's willingness to change project scope is a variable that increases with time. However, RealSoft was an ongoing project and more releases were in planning or under development. Both the simulated scope and the actual scope move upward again around day 120 (marked by arrow 2). A careful examination of the model behavior suggests that this is due to the dynamics of productivity (see Figure 2 in the online appendix). After a major refactoring done by day 100 in the project, design quality is much improved and the productivity increases for four iterations. The increased productivity causes an increase in the estimated number of tasks that can be finished.

In summary, the behavioral validation demonstrates that the SD simulation model developed in our research clearly predicts the dynamic behavior of critical project outcomes. In conjunction with the tests for structure validation, this behavioral validation establishes that the model is valid and very well reflects a typical agile software development project. Though none of these tests in isolation is sufficient for ensuring the validity of the model, in combination they provide a formidable filter to evaluate the performance of a model. The results from the well-established tests conducted in our study confirm that our model is sufficiently adequate and appropriate for use to investigate the dynamics of agile software development.

4. INSIGHTS FROM MODEL EXPERIMENTATION

After the model was validated, we conducted an experiment to examine two critical practices in agile software development. Specifically, we examined the role and impacts of refactoring on project performance and economics of pair programming using data from the RealSoft project. The RealSoft project was selected because it represented a typical medium-sized agile development project, following a hybrid methodology that uses practices from XP and SCRUM. The parameters of the model were set to values that reflected the RealSoft project environment. The SD model was specified in the STELLA® environment and its behavior was simulated to understand the dynamic effects of refactoring and pair programming. This experimentation illustrates the use of our SD simulation model as a "flight simulator" for agile development projects. IS project managers and project personnel can use our SD simulation to examine the impact of a variety of agile development practices and management policies as illustrated in the following subsections.

4.1 Refactoring and Its Impact

In real projects, the need for refactoring is not always obvious. Also, manually performing this task is fairly time consuming. Further, under schedule pressure, programmers will not perform refactoring [Roberts 1999]. Similarly, experienced programmers loathe modifying code that is working for fear of introducing unexpected errors. The aphorism "if it ain't broke, don't fix it" is the norm. Developers may get overly focused on delivering functionality and neglect critical needs for refactoring [Opdyke 1995]. As a result, actual refactoring effort is almost always less than the needed refactoring effort [Roberts 1999].

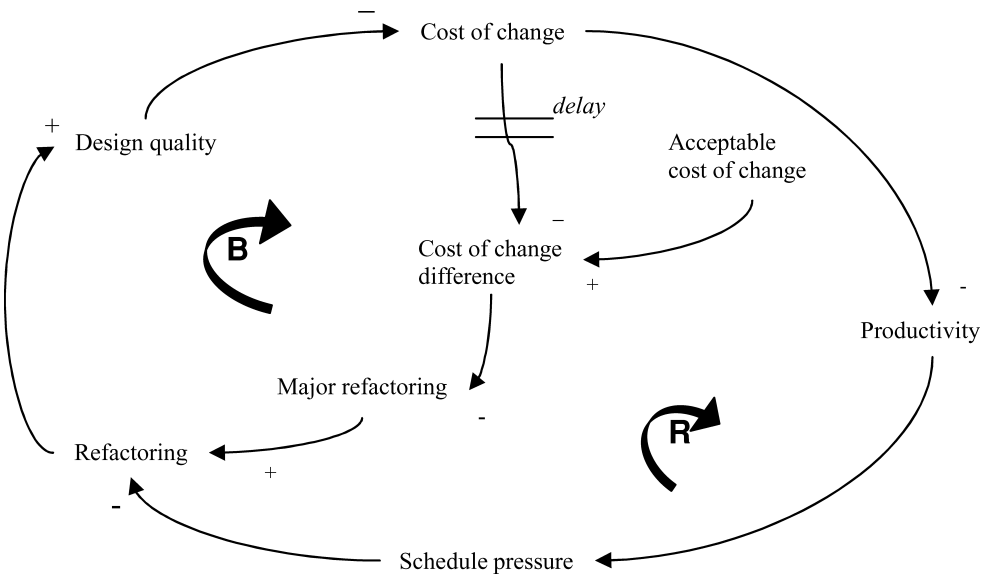


Fig. 7. Feedback structures of refactoring and design quality.

Figure 7 shows the feedback structures of refactoring and quality of design with a balancing loop (Loop B) and a reinforcing loop (Loop R). At the core is a balancing feedback loop with delay. Starting from the left, insufficient refactoring leads to poor design quality. Though project requirements change during development, the design is not updated immediately to accommodate new requirements. Changes are made in the most expedient way possible and the design of the system becomes increasingly ineffective and brittle. As design deteriorates, so does the ability to understand and evolve the code effectively. Not only does this make the software harder to change, but also it makes bugs easier to breed, as well as harder to detect and safely remove. Poorly designed code usually requires more effort to incorporate a new feature, and slows down development [Fowler 1999]. Therefore, the cost of making modifications increases. However, there is a threshold beyond which cost of modifications cannot be allowed to escalate. The development can proceed until this cost is below a threshold. Thus, there is a delay in perceiving the increased cost of making modifications caused by poor design quality until it reaches a point when the cost is above the threshold, or the delay in incorporating modifications is unacceptably long. Then, developers have to stop development and clean up the messy code (i.e., perform "major refactoring"). After major refactoring, the design is at an acceptable level of quality such that the developers can continue with another development cycle (completing/closing the balancing Loop B).

The more the design suffers from insufficient refactoring, the faster the decay in the quality of design. As a result, productivity declines and schedule pressure increases. In response, developers spend less time on refactoring and focus on finishing the development tasks assigned to them, which in turn

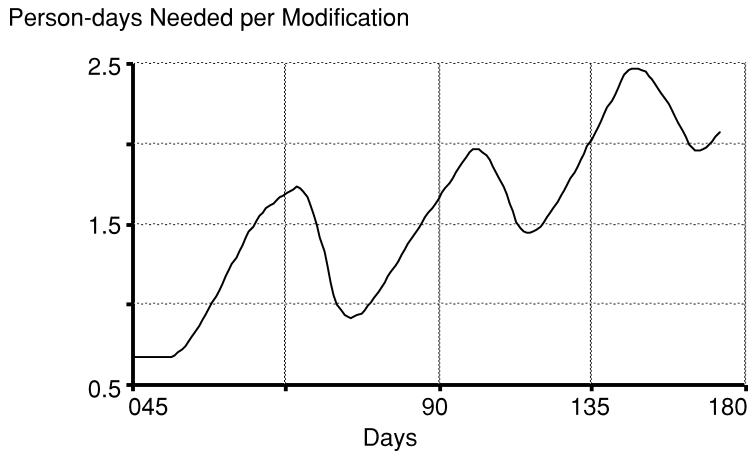


Fig. 8. Cost of change (person-days/change) in RealSoft project.

leads to further deterioration in design quality (completing the reinforcing Loop R).

Next, we discuss the impact of refactoring on cost of making modifications and the use of pair programming. It is long established that in traditional development the cost of modifying software increases exponentially with time [Boehm 1981]. Therefore, traditional methods emphasize the importance of accurate requirements and detailed up-front design. In agile development, the customer provides continuous feedback [Beck 2000] and the features that needed to be modified are identified as soon as they are developed. Resulting changes are likely to be small and inexpensive to implement, and as a result, it is claimed that the cost of making modifications is constant instead of increasing exponentially [Beck 2000]. However, since this claim has not been adequately verified by prior research, we examine it with our SD simulation of the RealSoft project.

The cost-of-change curve produced by the model is shown in Figure 8. It shows the person-days needed for each change. The cost per change increases over time and more interestingly, the curves go up and down in several cycles. Further, we find that the cost of change increases quickly within each small cycle. This increase is directly related to the need for refactoring. Ideally, developers should keep refactoring the code to maintain the quality of design. However, a lot of factors, such as schedule pressure, degree of pair programming, and most importantly, the lack of motivation, affect the actual degree of refactoring. Insufficient refactoring will decrease the quality of the design, and therefore accommodating change requests gets difficult over time, until developers start cleaning up the messy code by engaging in a significant refactoring exercise. Design is improved dramatically by significant refactoring and then the cost per change drops. However, the need for another cycle of refactoring begins right after a major refactoring activity. Figure 9 shows how design quality varies over time with insufficient refactoring.

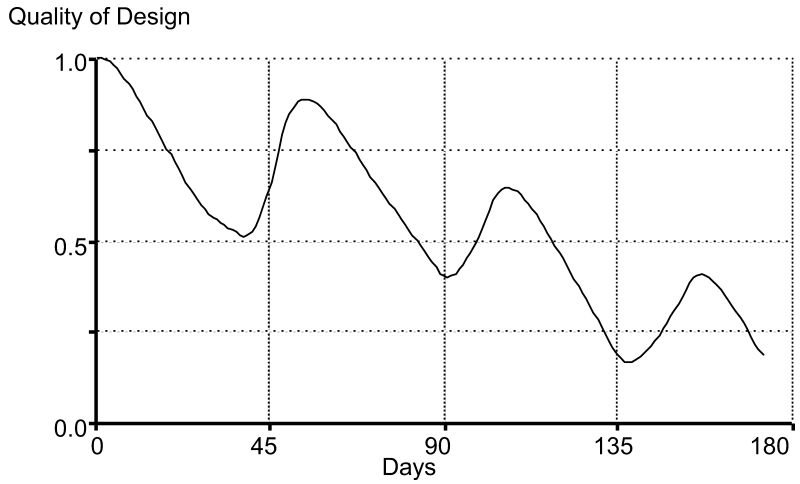


Fig. 9. Quality of design in RealSoft project.

The cost per change (person-days needed per change) ranges between 0.65 and 2.47 person-days. Our experiment suggests that in a typical agile project: (1) cost of change is not constant, but rather, its value changes cyclically; (2) overall, the cost of change increases over time but to a much lesser degree (4:1) than in traditional approach (100:1). The result has significant implication for agile methods: the smaller range of cost of change appears to make iterative development feasible. However, its applicability largely relies on the refactoring practice. Even though short iterations and frequent releases reduce the magnitude of changes dramatically, the cost of change can increase quickly if the design is not cleaned up regularly. This reveals a potential risk in agile development: if refactoring is not carefully done, the development process will be ineffective.

Next, we test the impact of refactoring behavior on development cost. Major refactoring is triggered when design quality drops to a level such that the effort needed for making a change (such as adding a new feature or modifying developed tasks) is much more than the normal effort. The developers recognize this problem, and then begin to clean up the system, which starts major refactoring. Figures 10 and 11 show how the delay in refactoring impacts project performance. The delay is represented by the value of quality of design when major refactoring is triggered. Quality of design is modeled as a variable with values ranging from zero (representing situations in which no major refactoring is done at all) to one (representing situations in which the code is continuously refactored). Figures 10 and 11 clearly show that project performance is very sensitive to the delay in major refactoring. In a project with a fixed schedule, more delay causes higher cost and fewer delivered stories. It is interesting to find that when delay is more than a certain threshold (in this case quality of design $= 0.4$), the increased delay results in lower costs and more delivered stories. The reason might be that the extra effort spent on major refactoring is not cost effective if it is done after much delay. However, without any

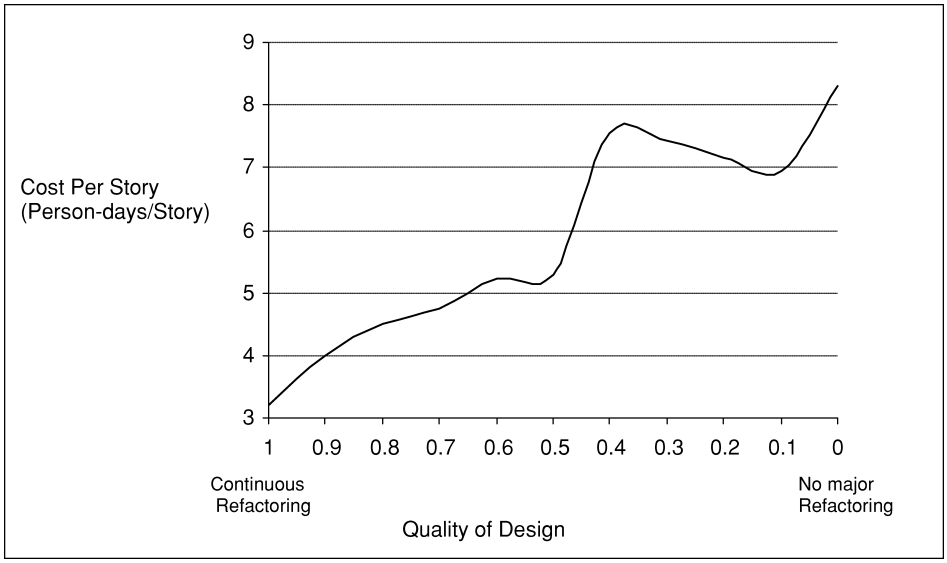


Fig. 10. Impact of delay of major refactoring on cost per story in RealSoft project.

major refactoring (quality of design close to zero), the project delivers the fewest stories at the highest cost per story.

The simulation shows that, in general, the higher the level of refactoring, the lower the overall development cost. However, since projects almost always suffer from insufficient refactoring, major refactoring is unavoidable. In such situations, development teams should plan for major refactoring in the release plan. Figures 10 and 11 show that major refactoring should be planned frequently. After the design quality drops beyond a minimally acceptable level, the cost of adding new features increases sharply.

In summary, refactoring is critical to agile software development, and project performance is very sensitive to delays in major refactoring.

4.2 Economics of Pair Programming

Studies have shown that pair programming results in less defects in code and higher speed than solo programming [Williams et al. 2000]. However, pair programming allocates two persons to each task, which increases labor costs. We use the SD simulation model of the RealSoft project to investigate the economics of pair programming.

We compare the results of two situations: (1) with 100% pair programming, and (2) no pair programming. The impacts on a variety of variables such as total cost of the project, cost of rework, cost of change, tasks delivered, and cost per task are shown in Table III.

With no pair programming, fewer tasks are delivered than with 100% pair programming. The cost per task delivered with no pair programming is 10% higher than that with pair programming. We compared the cost of rework and making changes with and without pair programming. The cost of rework

Table III. Comparing Pair Programming with No Pair Programming

	No Pair Programming	100% Pair Programming
Total cost (person-days)	848	853
Cost of rework (person-days)	120	88
Cost of change (person-days)	96	87
Tasks delivered	166	182
Cost of refactoring (person-days)	306	273
Cost per task (person-days)	5.17	4.68

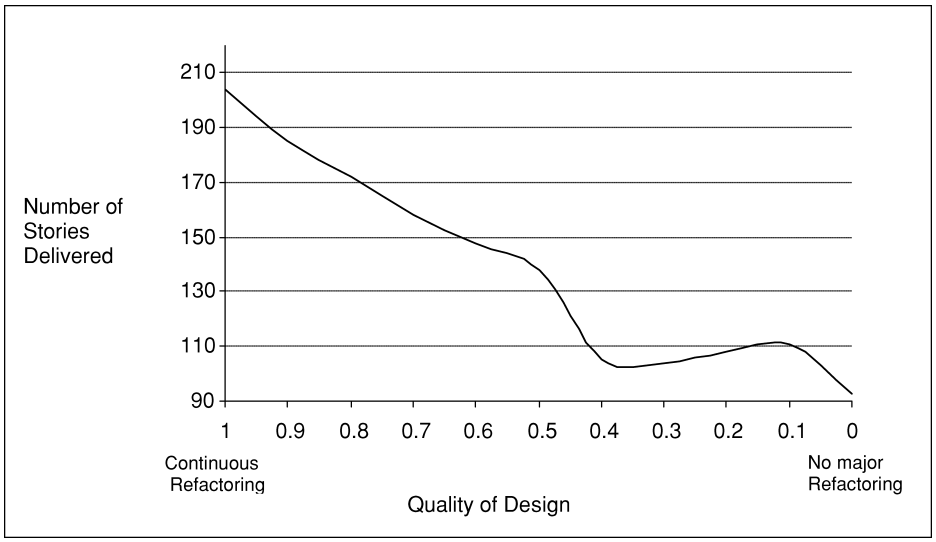


Fig. 11. Impact of delay of major refactoring on number of stories delivered in RealSoft project.

with no pair programming is 36% higher than that with pair programming. This results from the lower defects rate with pair programming. Also, with no pair programming, the cost of making changes is 10% higher and the cost of refactoring is 12% higher. This explains why the cost of delivering a task by paired teams is less than that without paired teams.

This experiment was conducted by keeping the degree of pair programming constant over time. In reality, the degree of pair programming is impacted by schedule pressure. The model accommodates this possibility as well. Generally speaking, in the simulated project, pair programming not only results in lower costs per task delivered but also increases the number of tasks delivered.

The behavior of the project with refactoring is examined to investigate the influence of pair programming. Figure 12 shows the effect of refactoring activities with and without pair programming. The actual minor refactoring effort with pair programming is higher than that with no pair programming. As a result, less "technical debt" is accumulated over time and only two major refactoring efforts are undertaken. The first happens around day 50 with pair programming compared to day 40 without it. Also, without pair programming this project requires three major refactoring activities.

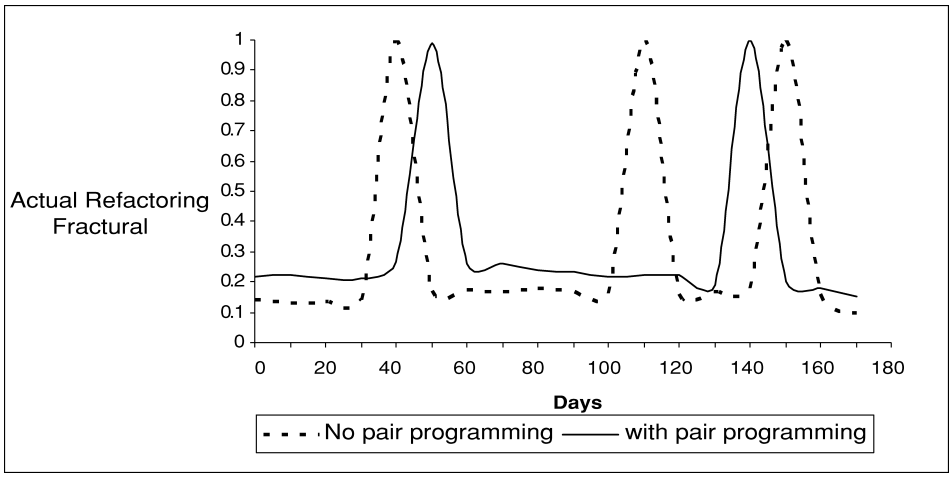


Fig. 12. Refactoring fractural with and without pair programming in RealSoft project.

Figure 12 clearly shows the impact of pair programming on refactoring behavior. With pair programming, the developers are more diligent in refactoring their code, and as a result the effort needed for major refactoring is less. Major refactoring is more costly than minor refactoring because "technical debt" increases the costs. Thus the total cost of refactoring is less with pair programming. Moreover, the need for fewer pauses in development due to major refactoring makes a smoother development process.

The model shows that pair programming results in higher total development cost. Also, some of the benefits of pair programming are realized through other practices such as refactoring. If these practices are enforced by certain mechanisms such as planned "hardening" iterations in RealSoft, then the benefit of pair programming might be less significant.

In summary, the experimentation with the SD simulation provides valuable insights on how two agile development practices affect critical project outcomes and behaviors. Similarly, the SD simulation model presented here can be used to examine the impacts of other agile development practices and managerial policies.

5. CONCLUSIONS

The objective of this research is to answer important questions of IS organizations concerning the use of agile methods. This research enhances our understanding of agile software development, especially the dynamic nature of agile practices when viewed as an integrated system. We have developed an integrative system dynamics model of agile software development, validated its structure and behavior, and used it to study the dynamic implications of two important practices (refactoring and pair programming) using a case study of a real project. The integrated model describes the behavior generated by the interaction of agile practices and project management including project scope, schedule, and cost.

The contributions of the research include the development of a new research tool to understand and analyze the impacts of agile development practices. Specifically, this tool helps investigate the following critical areas in agile development: customer involvement, change management, refactoring, agile planning and control, pair programming, and other commonly used agile practices. This research contributes to the literature on software development by providing a mechanism to study agile development as a dynamic system of practices rather than with a static view and in isolation. The model itself provides several building blocks that can be used in future research. Finally, the results from this study are expected to be of significant interest to IS organizations and practitioners of agile practices. This research provides them a simulation environment to examine the impact of their practices and policies.

Important feedback structures in agile software development are identified. The model behavior clearly shows that these feedback structures indeed determine important behaviors in agile development. The complex interactions between elements in agile software development are explicitly identified and represented in the model. Therefore, the model can be considered as a "theory" of agile software development and can be used to guide future research in this area. Our research investigates several claims about agile projects such as potential trade-offs among performance measures and reveals new insights about the dynamics of agile projects. For example, the model can be used by organizations to investigate the impacts of specific practices such as pair programming on project performance. The model can also be used to design and analyze project management policies.

An experiment with the model suggests that refactoring impacts the cost of making changes to the system. The cost changes cyclically and increases over time. This pattern is the result of refactoring behavior which is critical to the success of agile development. Also, in a project with a fixed schedule, delays with major refactoring lead to higher project cost and fewer delivered stories.

The impacts of pair programming on productivity, rework, refactoring, and unit testing were studied. The results show that with no pair programming, fewer tasks are delivered than with pair programming. The cost for each task delivered and the cost of rework and change with no pair programming are much higher than with pair programming. With pair programming, the need for major refactoring is less and the total cost of refactoring is reduced.

The model can help practitioners improve software development practice by facilitating the understanding of agile project dynamics. First, the model can be used to investigate the impacts of specific practices such as level of customer involvement on project performance. Second, the model can be used to design and analyze project management policies. For example, effort allocation between refactoring and development can be explored to find the optimal level of refactoring effort. Third, the model can be used to examine the sensitivity of agile development process to a variety of internal and external factors. For example, the sensitivity of agile process to team size, length of iteration duration, and the requirement volatility can be examined using the model. In summary, this research provides researchers a tool to study agile development, and helps practitioners make better decisions and formulate appropriate software processes.

REFERENCES

ABDEL-HAMID, T. K. 1989. The dynamics of software project staffing: A system dynamics based simulation approach. *IEEE Trans. Softw. Engin. 15*, 109–119.

ABDEL-HAMID, T. K. 1990. Investigating the cost/schedule trade-off in software development. *IEEE Softw. 7*, 97–105.

ABDEL-HAMID, T. K. 1993a. Adapting, correcting, and perfecting software estimates: A maintenance metaphor. *IEEE Comput. 26*, 20–29.

ABDEL-HAMID, T. K. 1993b. Modeling the dynamics of software reuse: An integrating system dynamics perspective. In *Proceedings of the 6th Workshop on Institutionalizing Software Reuse*.

ABDEL-HAMID, T. K. AND LEIDY, F. H. 1991. An expert simulator for allocating the quality assurance effort in software development. *Simul. 56*, 233–240.

ABDEL-HAMID, T. K. AND MADNICK, S. E. 1983. The dynamics of software project scheduling. *Comm. ACM 26*, 340–346.

ABDEL-HAMID, T. K. AND MADNICK, S. E. 1991. *Software Project Dynamics: An Integrated Approach*. Prentice Hall, Englewood Cliffs, NJ.

ABDEL-HAMID, T. K., SENGUPTA, K., AND HARDEBECK, M. J. 1994. The effect of reward structures on allocating shared staff resources among interdependent software projects: An experimental investigation. *IEEE Trans. Engin. Manag. 41*, 115–125.

ABDEL-HAMID, T. K., SENGUPTA, K., AND RONAN, D. 1993. Software project control: An experimental investigation of judgment with fallible information. *IEEE Trans. Softw. Engin. 19*, 603–612.

ALSHAYEB, M. AND LI, W. 2005. An empirical study of system design instability metric and design evolution in an agile software process. *J. Syst. Softw. 74*, 269–274.

ARNOLD, R. S. 1986. An introduction to software restructuring. In *Tutorial on Software Restructuring*, R. S. Arnold Ed., IEEE Computer Society Press, Los Alamitos, CA.

BARLAS, Y. 1989. Multiple test for validation of system dynamics type of simulation models. *Euro. J. Oper. Res. 42*, 59–87.

BASILI, V. R. AND WEISS, D. M. 1984. A methodology for collecting valid software engineering data. *IEEE Trans. Softw. Engin. 10*, 728–737.

BECK, K. 1994. Simple smalltalk testing: With patterns. *Smalltalk Rep. 4*.

BECK, K. 2000. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston, MA.

BECK, K. 2003. *Test Driven Development: By Example*. Addison-Wesley Professional.

BECK, K. AND GAMMA, E. 1998. Test infected: Programmers love writing tests. *Java Rep. 3*, 37–50.

BERRY, D. M. 2002. The inevitable pain of software development, including of extreme programming, caused by requirements volatility. In *Proceedings of the International Workshop On Time-Constrained Requirements*.

BOEHM, B. 1981. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ.

BOEHM, B. 2002. Get ready for agile methods, with care. *IEEE Comput. 35*, 64–69.

BOEHM, B. AND TURNER, R. 2004. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison Wesley.

CANGUSSU, J. W. 2004. A software test process stochastic control model based on cmm characterization. *Softw. Process. Improv. Pract. 9*, 55–66.

CAO, L., MOHAN, K., XU, P., AND RAMESH, B. 2009. A framework for adapting agile development methodologies. *Euro. J. Inform. Syst. 18*, 332–343.

CHOI, S. J. AND SCACCHI, W. 2001. Modeling and simulating software acquisition process architectures. *J. Syst. Softw. 59*, 343–354.

CUNNINGHAM, W. 1992. The Wycash portfolio management system. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM Press, New York.

CURTIS, B., KRASNER, H., AND ISCOE, N. 1988. A field study of the software design process for large systems. *Comm. ACM 31*, 1268–1287.

DUTTA, A. AND ROY, R. 2005. Offshore outsourcing: A dynamic causal model of counteracting forces. *J. Manag. Inform. Syst. 22*, 15–35.

DUTTA, A. AND ROY, R. 2008. Dynamics of organizational information security. *Syst. Dynam. Rev. 24*, 349–375.

ELSSAMADISY, A. AND SCHALLOIL, G. 2002. Recognizing and responding to 'bad smells' in xp. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*.

ERDOGMUS, H. AND WILLIAMS, L. 2003. The economics of software development by pair programmers. *Engin. Econom. 48*, 283–319.

ERICKSON, J., LYYTINEN, K., AND SIAU, K. 2005. Agile modeling, agile software development, and extreme programming: The state of research. *J. Data. Manag. 16*, 88–99.

FITZGERALD, B., HARTNETT, G., AND CONBOY, K. 2006. Customizing agile methods to software practices at Intel Shannon. *Euro. J. Inform. Syst. 15*, 200–213.

FORRESTER, J. W. 1961. *Industrial Dynamics*. MIT Press, Cambridge, MA.

FORRESTER, J. W. AND SENGE, P. M. 1980. Tests for building confidence in system dynamics models. *TIMS Stud. Manag. Sci. 14*, 209–228.

FOWLER, M. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

GEORGE, B. AND WILLIAMS, L. 2002. An initial investigation of test-driven development in industry. In *Proceedings of the ACM Symposium on Applied Computing*.

GLASS, R. 2004. Matching methodology to problem domain. *Comm. ACM 47*, 19–21.

GORTS, S. 2004. Refactoring in the large. http://www.refactoring.be/thumbnails/large/large.html

GRENNING, J. 2001. Launching xp at a process-intensive company. *IEEE Softw. 18*, 3–9.

HARRISON, B. 2003. A study of extreme programming in a large company. http://www.agilealliance.org/articles/articles/ALR-2003-039-paper.pdf

HIGHSMITH, J. AND COCKBURN, A. 2001. Agile software development: The business of innovation. *IEEE Comput. 34*, 120–122.

HODGETTS, P. 2004. Refactoring the development process: Experiences with the incremental adoption of agile practices. In *Proceedings of the IEEE Agile Development Conference*.

HÖST, M., REGNELL, B., DAG, J. N. O., NEDSTAM, J., AND NYBERG, C. 2001. Exploring bottlenecks in market-driven requirements management processes with discrete event simulation. *J. Syst. Softw. 59*, 323–332.

JEFFRIES, R., ANDERSON, A., AND HENDRICKSON, C. 2001. *Extreme Programming Installed*. Addison-Wesley, Boston, MA.

KARLSTRÖM, D. 2002. Introducing extreme programming—An experience report. In *Proceedings of the 3rd International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP'02)*.

KATAOKA, Y., IMAI, T., ANDOU, H., AND FUKAYA, T. 2002. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the International Conference on Software Maintenance*. 576–585.

LI, W. AND HENRY, S. 1993. Object-Oriented metrics that predict maintainability. *J. Syst. Softw. 23*, 111–122.

LIENTZ, B. P., SWANSON, E. B., AND TOMPKINS, G. E. 1978. Characteristics of application software maintenance. *Comm. ACM 21*, 466–471.

MADACHY, R. 2007*Software Process Dynamics . Wiley-IEEE Press, Los Alamitos, CA.*

MARTIN, R. C. *2000. Extreme programming development through dialog. IEEE Softw. 17, 12–13.*

MENS, T. AND TOURWE, T. *2004. A survey of software refactoring. IEEE Trans. Softw. Engin. 30, 126–139.*

NOSEK, J. T. *1998. The case for collaborative programming. Comm. ACM 41, 105–108.*

OPDYKE, W. F. *1992. Refactoring: A program restructuring aid in designing object-oriented application frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign.*

OPDYKE, W. F. *1995. Refactoring object-oriented software to support evolution and reuse. In Proceedings of the 7th Annual Workshop on Institutionalizing Software Reuse.*

PARRISH, A., SMITH, R., HALE, D., AND HALE, J. *2004. A field study of developer pairs: Productivity impacts and implications. IEEE Softw. 21, 76–79.*

PETERS, P. AND JARKE, M. *1996. Simulating the impact of information flows in networked organizations. In Proceedings of the International Conference on Information Systems (ICIS).*

RAMESH, B., CAO, L., AND BASKERVILLE, R. *2010. Agile requirements engineering practices and challenges: An empirical study. Inform. Syst. J. 20, 5, 449–480.*

ROBERTS, D. *1999. Practical analysis for refactoring. Ph.D. thesis, University of Illinois at Urbana Champaign.*

ROEHLING, S. T., COLLOFELLO, J. S., HERMANN, B. G., AND SMITH-DANIELS, D. E. *2000. System dynamics modeling applied to software outsourcing decision support. Softw. Process. Improv. Pract. 5.*

RUMPE, B. AND SCHROEDER, A. *2002. Quantitative survey on extreme programming project. In Proceedings of the 3rd International Conference on eXtreme Programming and Agile Processes in Software Engineering. 95–100.*

RUS, I., COLLOFELLO, J., AND LAKEY, P. *1999. Software process simulation for reliability management. J. Syst. Softw. 46, 173–182.*

SENGUPTA, K. AND ABDEL-HAMID, T. K. *1996. The impact of unreliable information on the management of software projects: A dynamic decision perspective. IEEE Trans. Syst. Man Cybernet. 26, 177–189.*

SENGUPTA, K., ABDEL-HAMID, T. K., AND BOSLEY, M. *1999. Coping with staffing delays in software project management: An experimental investigation. IEEE Trans. Syst. Man Cybernet. A29, 77–91.*

SHORE, J. *2004. Continuous design. IEEE Comput. 21, 20–22.*

STALLINGER, F. AND GRÜNBACHER, P. *2001. System dynamics modeling and simulation of collaborative requirements engineering. J. Syst. Softw. 59, 311–321.*

STARK, G. E., OMAN, P., SKILLICORN, A., AND AMEELE, A. *1999. An examination of the effects of requirements changes on software maintenance releases. J. Softw. Maint. Res. Pract. 11, 293–309.*

STERMAN, J. D. *2000. Business Dynamics: System Thinking and Modeling for a Complex World. McGraw-Hill.*

STROULIA, E. AND LEITCH, R. *2003. Understanding the economics of refactoring. In Proceedings of the 5th ICSE Workshop on Economics-Driven Software Engineering Research.*

SWANSON, E. B. *1976. The dimensions of maintenance. In Proceedings of the 2nd Conference on Software Engineering. 492–497.*

WEST, D., GRANT, T., GERUSH, M., AND D'SILVA, D. *2010. Agile Development: Mainstream Adoption Has Changed Agility. Forrester Research.*

WHILER, O. *2003. Refactoring. Methods & Tools Spring.*

WILLIAMS, L. *2001. Integrating pair programming into a software development process. In Proceedings of the 14th Conference on Software Engineering Education and Training.*

WILLIAMS, L. AND KESSLER, R. R. *2000. The effects of "pair-pressure" and "pair-learning" on software engineering education. In Proceedings of the 13th Conference of Software Engineering Education and Training.*

WILLIAMS, L., KESSLER, R. R., CUNNINGHAM, W., AND JEFFRIES, R. *2000. Strengthening the case for pair-programming. IEEE Softw. 17, 19–25.*

Received February 2010; revised September 2010; accepted September 2010