TEAM-RUP: AN AGENT-BASED SIMULATION STUDY OF TEAM BEHAVIOR

IN SOFTWARE DEVELOPMENT ORGANIZATIONS

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisory committee. This thesis does not include proprietary or classified information.

_____

Jared Phillips

Certificate of Approval:

_____	_____
Hari Narayanan Associate Professor Computer Science & Software Engineering	Levent Yilmaz, Chair Assistant Professor Computer Science & Software Engineering
_____	_____
David Umphress Associate Professor Computer Science & Software Engineering	Stephen McFarland Acting Dean Graduate School

TEAM-RUP: AN AGENT-BASED SIMULATION STUDY OF TEAM BEHAVIOR IN SOFTWARE DEVELOPMENT ORGANIZATIONS

Jared R. Phillips

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

Auburn, Alabama
May 11, 2006

TEAM-RUP: AN AGENT-BASED SIMULATION STUDY OF TEAM BEHAVIOR

IN SOFTWARE DEVELOPMENT ORGANIZATIONS

Jared R. Phillips

_____

Signature of Author

_____

Date of Graduation

THESIS ABSTRACT

TEAM-RUP: AN AGENT-BASED SIMULATION STUDY OF TEAM BEHAVIOR IN SOFTWARE DEVELOPMENT ORGANIZATIONS

Jared R. Phillips

Master of Science, May 11, 2006
(M.S., Emory University, 2003)
(B.S., University of Montevallo, 2002)

113 Typed Pages

Directed by Levent Yilmaz

Software production methods are enacted via the interactions of software teams that cooperate to build software. Therefore, organizational culture has a significant effect on project coordination. Yet, this is not reflected in current software process simulation efforts. This thesis introduces a new simulation model development framework, called Team-RUP, to facilitate exploration of the effects of team behavior on the efficiency and effectiveness of software development organizations that pursue incremental and iterative processes such as the Rational Unified Process (RUP). Team-RUP organizes teams according to the degree of autonomy in collaboration and the degree of concurrency in coordination, resulting in four distinct team archetypes: Autonomous, Agile, Concurrent, and Synchronized. Each team archetype embodies a unique combination of autonomy and concurrency levels, highly reflective of modern organizational paradigms. Using Team-RUP, we explore the effectiveness and efficiency of team archetypes under various

levels of task complexity and stability (i.e., internal and external turbulence), as well as team size and workload factors. The conclusions of the simulation study support the claim that process agility is a valid and useful counterbalance to the inevitable change involved in most real-world software projects. In particular, small organizations should consider adopting a software process that encourages agile behavior. If greater independence among teams is necessitated by a particular project, a large organization will perform significantly better than a smaller one.

ACKNOWLEDGEMENTS

The author would like to thank Dr. Levent Yilmaz for his guidance as well as other members of his thesis committee: Dr. Hari Narayanan and Dr. David Umphress. In addition, he appreciates the valuable input offered by the simulation research groups in both the Department of Computer Science & Software Engineering as well as the Department of Industrial & Systems Engineering. Finally, a special thanks is owned to the author's wife Rebecca Phillips and his family for their continued support.

Style Manual Used: Publication Manual of the American Psychological Association

Computer Software Used:

- Repast Agent Simulation Toolkit
- Microsoft Word 2003
- Microsoft Excel 2003
- Microsoft Visio 2003
- Microsoft PowerPoint 2003
- Java 1.5.0

TABLE OF CONTENTS

LIST OF FIGURES x
LIST OF TABLES xi
CHAPTER 1 1
Introduction 1
CHAPTER 2 7
Software Process Simulation: Making the Case for 7
the Rational Unified Process 7
1. Software Process Simulation 7
1.1 Analytic Models 8
1.2 System Dynamics 8
1.3 Discrete Models 9
1.4 Hybrid Models 10
1.5 Agent-Based Models 10
2. Rational Unified Process 12
2.1 Definition & History 12
2.2 Static Structure 13
2.3 Dynamic Structure 14
CHAPTER 3 19
Conceptualization of Team-RUP: 19
Organization, Task, and Performance Models 19
1. Organizational Model 20
1.1 Synchronized Teams 21
1.2 Concurrent Teams 22
1.3 Agile Teams 22
1.4 Autonomous Teams 22
1.5 Emergent Behavior 23
2. Task Model 24
2.1 Project Configuration 25
2.2 Risk Management 26
2.3 Shell Sort 28
3. Performance Model 31
3.1 Software Metrics 31
CHAPTER 4 33
The Design and Implementation of Team-RUP 33
1. High Level Overview 33
2. Team Types via Sorting Algorithms 35
2.1 Autonomous Teams via Merge Sort 36

2.2 Concurrent Teams via Quick Sort....38
2.3 Agile Teams via Insertion Sort....39
2.4 Synchronized Teams via Heap Sort....41
2.5 Incorporating the Experience Factor into the Model....43
3. Design and Implementation....44
CHAPTER 5....49
Turbulence....49
CHAPTER 6:....57
Programmed Model....57
1. Deadlines & Timing....57
2. Repast....58
3. Verification....58
4. Validation....61
CHAPTER 7:....69
Experimental Model....69
1. Agility Test....69
1.1. Experiment Design....70
1.2 Results....71
1.3 Discussion....73
2. Team Size Test....75
2.1. Experiment Design....75
2.2 Results....75
2.3 Discussion....80
CHAPTER 8:....84
Conclusion....84
1. Result Summary....85
2. Future Work....86
REFERENCES....89
APPENDICES....95
APPENDIX A....96
APPENDIX B....97

LIST OF FIGURES

Figure 1.1. The Subject Area of Team-RUP....2
Figure 2.1. Taxonomy of Simulation Techniques....11
Figure 2.2. The RUP lifecycle (adapted from Larman, 2004; Kruchten, 2003)....15
Figure 3.1. Realization of the Conceptual Model....20
Figure 3.2. The Structure of an Organization in the Team-RUP Framework....20
Figure 3.3: Team Behaviors....23
Figure 4.1. Iteration Workflow for Autonomous Teams....34
Figure 4.2. Operation of Autonomous Teams....36
Figure 4.3. Operation of Autonomous Teams (continued)....37
Figure 4.4. Operation of Concurrent Teams....38
Figure 4.5. Operation of Concurrent Teams (continued)....39
Figure 4.6. Operation of Agile Teams....40
Figure 4.7. Operation of Agile Teams (continued)....41
Figure 4.8. Operation of Synchronized Teams....42
Figure 4.9. Operation of Synchronized Teams (continued)....43
Figure 4.10. Operation of Synchronized Teams (continued)....43
Figure 4.11. UML Package Diagram for the Overall Design....45
Figure 4.12. Content of edu.auburn.philljr.team package....46
Figure 5.1. Ambiguous Inversion Removal of (4,3)....50
Figure 6.1. Graphical Depiction of List Inversions....59
Figure 6.2. Predictability of Team Productivity....63
Figure 6.3. Predictability of Staff Utilization....63
Figure 6.4. Predictability of Timeliness....64
Figure 6.5. Predictability of Quality....64
Figure 6.6. Quality for Agile, Autonomous, and Concurrent Teams....66
Figure 6.7. Quality for Synchronized Teams....67
Figure 6.8. Inefficiency of Agile Teams....68
Figure 7.1. Productivity Under Various Turbulence Levels....71
Figure 7.2. Staff Utilization Under Various Turbulence Levels....72
Figure 7.3. Timeliness Under Various Turbulence Levels....72
Figure 7.4. Quality Under Various Turbulence Levels....73
Figure 7.5. Series Legend for Results in Section 2.2....75
Figure 7.6. Productivity for Team Sets of Various Sizes....76
Figure 7.7. Staff Utilization for Team Sets of Various Sizes....77
Figure 7.8. Timeliness for Team Sets of Various Sizes....78
Figure 7.9. Quality for Team Sets of Various Sizes....79
Figure A.1. Team-RUP Design Class Diagram....96

LIST OF TABLES

Table 6.1. Factor Coding, coordination validation experiment 62
Table 6.2. Factor Coding, collaboration validation experiment.....66
Table 7.1. Level Definitions for Internal and External Turbulence 70
Table 7.2. Level Definitions for Turbulence.....70
Table 7.3. Level Definitions for Number of Teams 75
Table B.1. Factor Levels for the Agility Test 97
Table B.2. Response Values for the Agility Test.....98
Table B.3. Factor Levels for the Team Size Test.....100
Table B.4. Response Values for the Team Size Test 102

CHAPTER 1

Introduction

Software process simulation is a well established, albeit underused, technique for improving software development. It facilitates prediction capabilities, employee training, and improvement of tailored processes. Simulation can also be utilized in broadening the collective understanding of more generic processes and archetypical scenarios. Rather than matching data collected from real-world enterprises, models used in this latter form of simulation capture high-level properties that provide insight into underlying organizational trends. Simulations of this type are reminiscent of Axelrod's (1997) tribute model in the realm of social science. The Team-RUP study falls into this latter category. As depicted in Figure 1.1, Team-RUP is a multi-faceted framework for exploring the diverse problem set associated with the intersection of computer science, sociology, and management science.

Large-scale development of software involves multiple groups of collaborating professionals collectively influenced by team dynamics: the expertise of sociologists. To cope with the complexity of software engineering, developers have borrowed ideas from management science to create formal processes to ensure a measure of conformity and stability among teams. Such processes, however, have not defined a single, archetypal software development team. Rather, multiple organizational paradigms have emerged

characterized by the manner in which team members interact, decompose work, and assemble a final product.

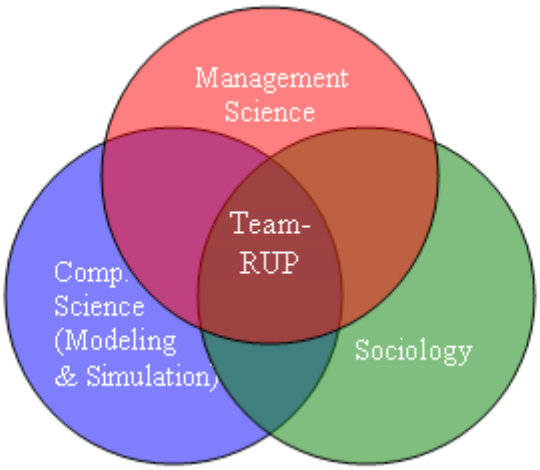

Figure 1.1. The Subject Area of Team-RUP

The intricacy resulting from this confluence of disciplines is further augmented by the unique nature of software projects. Software engineers develop a product constantly subject to requirements change. Moreover, software is not bound to the physical world. It is a symbolic record of the abstract thoughts of its creators. This latter fact elevates the significance and complexity of communication among teams. For these reasons, an understanding of the behavior of software development teams cannot be obtained deductively from knowledge of the three disciplines shown in Figure 1.1.

Software team behavior is a unique domain of phenomena that must be studied separately from other products of human interaction. Much of the work in this area derives from personal anecdotes and features useful, albeit subjective, experience-based heuristics. The few empirical studies of this behavior indicate a vast discrepancy between the perceived and actual nature of software development teams. For example, Sawyer and Guinan (1998) have provided evidence that social processes have a greater

effect than production methods on the quality of software created by a team. An example germane to Team-RUP is research performed by Larry Constantine (1993). Constantine mapped software team behaviors onto a two-dimensional spectrum that places four archetypal organizational paradigms at the diagonal extremities. Each paradigm is the limiting behavior observed as a team progresses along one of four axes: divergence, reflexivity, hierarchy, and alignment. The author also provides lists of properties that characterize teams belonging to these paradigms (Constantine, 1993). As is characteristic of all research bound to irreproducible historical events, these studies lack several desirable features that can be obtained through simulation.

Simulation is a tool that allows a researcher to consider a problem from a variety of viewpoints. In addition to recreating historical events, simulation is useful for predicting future events, performing "what-if" and trend analysis, and for problem modification. One can filter out the variability due to exceptional circumstances to derive more general, far-reaching results. These benefits motivate the development of a new simulation framework to address the limitations of empirical research as it pertains to software process simulation. Team-RUP is a simulation framework designed for the mutual study of software development, team dynamics, processes, organizational paradigms, and cultures. Its uses include the validation of empirical studies, the discovery of general trends among teams in varying environments, and the identification of causal relationships between management decisions and team performance.

Team-RUP bases its theme on the Team-Soar project. This latter study focused on decision strategies used by a command and control team consisting of ships, aircraft, and a land-based unit (Kang, Waisel, and Wallace, 1998). Researchers were able analyze the

effects of various voting rules that give different priorities to majority opinions and team member knowledge levels. Team-RUP is also a team-centric study. In contrast to Team-Soar, however, the "Team" is the software construction division of a software development company. Its members consist of a project manager, a design manager, and a variable number of construction teams. Rather than investigating decision making, Team-RUP is directed at understanding the effects of various team behaviors on the efficiency and effectiveness of the organization.

The use of simulation to study interactions between software developers and team behavior is not a new idea. For example, Raffo, Setamanit, and Wakeland (2003) combined two well-established simulation techniques, system dynamics and discrete-event, to study processes governing globally distributed software teams. Lucas and Goss (1999) employed software agents to study team behavior in military applications. Falling closest to Team-RUP is the work of Wickenberg and Davidsson (2003). They advocate the use of agents to study software processes and team behavior. Team-RUP, however, is not a rehashing of old ideas.

It represents a new and unique research tool for several reasons. First, it focuses on one widely used process framework: RUP. While its intent is to provide general results, Team-RUP possesses enough grounding to be useful in a real-world context. It also utilizes a distinctive layered approach allowing features to be modeled at the individual, team, and enterprise levels. Moreover, cross-cutting factors like internal turbulence can be captured. Team-RUP presents a new modeling strategy based on sorting. Finally, it uses inter-agent communication as a central modeling component rather than as a support tool for agent collaboration.

Team-RUP approaches the problem of classifying team behaviors from an agent-oriented perspective that differs significantly from Constantine's (1993) methodology. In particular, the Team-RUP taxonomy organizes teams according to the degree of autonomy in collaboration and the degree of concurrency in coordination. The four combinations in this latter cataloging scheme, however, exhibit characteristics similar to the former. This research, therefore, provides the opportunity to further explore a well-established framework using computational methods. Using simulation, the move can be made beyond identification to implication. Using the Team-RUP framework, we can predict general trends in the efficiency and effectiveness of teams exhibiting certain behaviors in particular situations. To achieve this end, the Team-RUP study progressed through several milestones.

The tasks involved in the Team-RUP study fall into three broad categories. First, a conceptual framework was chosen that provides a level of specification detailed enough to bind the simulation model to real-world software development. As evinced by its name, Team-RUP anchors itself by conforming to the principle tenets of the Rational Unified Process. A metaphor based on list sorting was selected to simulate software construction and the various team behaviors. Next, an agent-based model was constructed combining elements of the event scheduling and activity scanning simulation worldviews (Banks, 1998; Banks, Carson, Nelson, and Nicol, 2005). Validation experiments were designed based on documented facts to test the operational validity of the model in terms of the coordination and collaboration of teams. Because the Agile behavior is of particular interest to this study, an additional validation experiment was conducted centering on properties of this particular behavior.

The final phase of this study consisted of conducting experiments to investigate the effects that turbulence and the number of teams have on varying team behaviors. The results of these experiments allowed us to conclude that "agility" is a valid and useful counterbalance to the inevitable change involved in most real-world software projects. In particular, small organizations should strongly consider adopting a software process that encourages agile behavior. If, however, an Autonomous or Concurrent strategy is somehow necessitated by a particular project, a large organization will perform better than a smaller one.

CHAPTER 2

Software Process Simulation: Making the Case for the Rational Unified Process

As a vehicle for software process improvement, Team-RUP lies in the confluence of two streams of thought and perspective. The first is the use of simulation as a tool for the understanding and betterment of software processes. These simulation studies may exhibit significant academic value and often provide support for or suggest changes to currently used methodologies. The second sources of insight comes from a commercial venture; namely, the Rational Unified Process. RUP is a collection of methodologies for creating software whose survival in the marketplace has proven its value. The following two sub-sections are devoted to these two foundational elements.

1. Software Process Simulation

Little doubt remains over whether software processes are beneficial. When properly calibrated, a software process greatly aids developers in delivering a product that is high quality, on time, and at cost. A poorly calibrated process, however, can have the opposite effect. Moreover, initially implementing a process in the real world can require significant amounts of time and money. Simulating process decisions before they are actually implemented can alleviate the risk of failure. Simulation can also be used to improve processes already in place. With these benefits in mind, software engineers have

developed multiple simulation techniques, each of which has advantages, disadvantages, and optimal conditions for application.

1.1 Analytic Models

The idea of creating abstract models to better understand software processes is by no means new. Analytic models (Albrecht, 1979; Boehm, 1981) that find representation to this day trace their histories back to the late 1970's and early 1980's. These models consist of mathematical equations that represent relationships between various process entities. Although useful in certain contexts, analytic models divorce process attributes from their effects and fail to consider dynamic interaction of process factors (Donzelli and Iazeolla, 2001; Martin and Raffo, 2000). Simulation modeling has been used to overcome these deficiencies.

1.2 System Dynamics

The most practiced form of simulation modeling in the context of process simulation is system dynamics. System dynamics arose from Jay Forrester's research (1961) concerning the dynamics of business and social systems. It involves the modeling of systems in terms of feedback loops and ordinary differential equations. In the field of software process, a seminal work aimed at understanding human resource management, software production, controlling, and planning was developed by Abdel-Hamid and Stuart Madnick in 1991. Lehman and Ramil (1999) have used the system dynamics approach to provide support for Lehman's laws: a system of high-level propositions concerning software systems embedded and used in real-world settings. System dynamics models have even proven useful in the domain of software acquisition as

evidenced by the work of Haberlein (2004). A drawback of system dynamics, however, is the need for complete and precise empirical information to calibrate the ordinary differential equations (Ramil and Smith, 2002).

One solution to this problem that allows the researcher to remain in the realm of continuous dependent variables replaces the ODE's from system dynamics with qualitative differential equations. By showing that well-known quantitative models are embedded within qualitative models, Ramil and Smith lend support to the idea that qualitative simulation provides a natural abstraction of system dynamics. Another strategy for addressing the weaknesses of system dynamics models involves abandoning a continuous world-view in favor of discrete models.

1.3 Discrete Models

Discrete models have contributed greatly to the understanding of software processes. Discrete-event models, which utilize event-scheduling, process interaction, or three-phase simulation techniques, allow queues to be easily represented and can postpone processing until required resources become available (Martin and Raffo, 2000). For example, Padberg (2002) has used discrete-event modeling to study software project scheduling policies. Another benefit of this technique is that model entities can be described in detail via assigned attributes (Martin and Raffo, 2000). Raffo, Harrison, and Vandeville (2000) took advantage of this characteristic in designing a predictive framework for process models that links models to dynamically updated metric repositories. State-based modeling is another discrete approach that has proven useful. State-based models adeptly represent process-level details and are highly compatible with graphical representation (Raffo, Vandeville, and Martin, 1999). For example, Raffo et al. (1999) developed a

state-based model of the software lifecycle used by Northrop Grumman's Surveillance and Battle Management Systems Division to serve as a business case for high process maturity. Concrete examples of state-based models include Petri-net and cellular automata models.

1.4 Hybrid Models

Efforts have also been made to combine continuous and discrete models. Donzelli and Iazeolla (2001) created a hybrid model of a waterfall lifecycle in which high-level details were managed in a discrete-event fashion, whereas lower level aspects combined features of system dynamics and analytic models. Martin and Raffo (2000) developed a similar model in which a discrete approach was used to model software development steps, whereas the environment in which the steps were carried out varied continuously. This hybrid approach to simulation modeling, however, remains in its infancy. An area of process simulation with a similar level of maturity centers on agents.

1.5 Agent-Based Models

Agent-based simulation presents a wealth of possibility to the software process community. Unlike other simulation techniques, agent-based simulation allows societies to emerge within the technical process structure. In the words of Wickenberg and Davidsson (2003), agent-based simulation of processes provides "a natural way to describe both communication between individuals, individual characteristics, and the discrete decisions made during the negotiations" (p. 10). Strangely enough, research in this field is limited but not new. For example, Mi and Scacchi (1990) used agents to simulate processes more than a decade ago in their work developing the Articulator environment. Scacchi reports that interest in the agents themselves supplanted his

research group's interest in studying processes (Scacchi, 1999). Wickenberg and Davidsson (2003), however, have provided new impetus to the field by developing sufficiency tests for the usefulness of agent-based process simulation as opposed to other simulation techniques. As shown in Figure 2.1, the agent-based approach typically serves as a shell for one of the other simulation techniques.

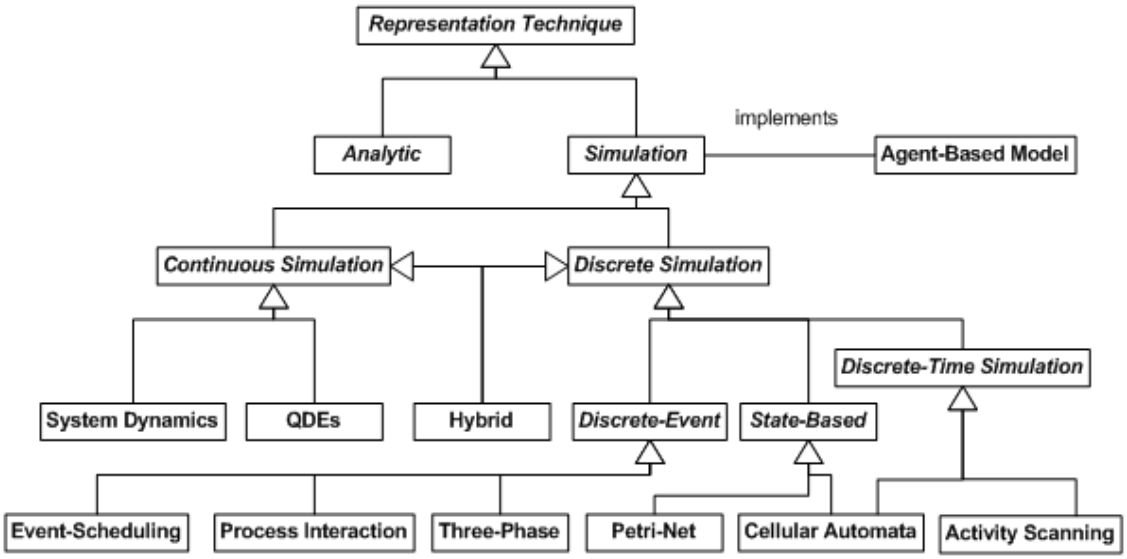

Figure 2.1. Taxonomy of Simulation Techniques

The UML (conceptual) class diagram in Figure 2.1 provides a simple taxonomy of representation techniques used to derive information about a system. Aside from the *implements* association between the Simulation and Agent-Based Model entities, the dependencies depicted in this figure are *is-a* relationships. For example, event-scheduling is a type of discrete-event simulation. Hybrid simulations combine continuous and discrete simulation techniques. Note that agent-based models are not bound to a particular low-level simulation technique. Instead, they are an abstraction that can be implemented in a variety of ways. Activity Scanning and State-Based approaches fit nicely with the agent-based paradigm.

2. Rational Unified Process

2.1 Definition & History

What RUP is and how it came to be are two inextricably bound concepts. This relationship results from the fact that RUP is both a software engineering process and a process product. That is, it was developed and is maintained in the same manner as every other software product. In particular, Rational Software Corporation has collected and integrated the best practices of the software development industry; in parallel, it has created a suite of tools to aid in the implementation of these practices. Over a period of ten years, the process and its progenitor meta-process have been expanded and honed (Krutchen, 2003).

Although Rational's association with the process spans a decade, the beginning of RUP can be found eight years earlier in 1987. Deriving from his work at Ericsson, Ivar Jacobson created Objectory process, which merged his earlier invention, the use case, with an approach to developing object-oriented software. This early ancestor of RUP became a product of Jacobson's company Objectory AB, which merged with Rational in 1995. This merger resulted in a hybridization of the Objectory process with the Rational Approach, an iterative technique for software development. Mergers with Requisite, Inc. and SQA, Inc. added requirements management and a testing process to the mixture. The adaptation of this combination to incorporate concepts from the Unified Modeling Language resulted in the creation of the Rational Objectory Process. A merger with Pure-Atria led to advances in configuration management and the first use of the name "Rational Unified Process" in 1998 (Krutchen, 1999).

Since this time, RUP has continued to evolve under a wide variety of environmental influences. Applying its iterative and incremental meta-process, Rational has continued to make adjustments to their product. Moreover, IBM acquired Rational in 2002 endowing RUP with access to one of the largest asset pools in the technology sector. Also, RUP has harvested ideas emerging from the parallel development of the Unified Process, a more general, public domain process (Larman, 2002). Finally, RUP has borrowed many ideas seeded in Agile community. For example, newer editions to RUP resemble methods found in Extreme Programming and Scrum (Pollice, Augustine, Lowe, and Madhur, 2004). This diversity of components alludes to a final face of RUP: process framework.

Because of its size and heterogeneity, RUP can be complicated and unwieldy if taken as a whole. For this reason, it should be viewed as a process framework that is tailored to meet the needs of its user. In other words, an organization should only use those parts (e.g., artifacts) of RUP that it needs. Otherwise, its use can be more of a hindrance than an aid (Probasco, 2001). For this very reason, RUP lists the Development Case among its key artifacts. This document details how an organization will adjust the RUP to suit its needs (Pollice, 2004). The Development Case is an element of RUP's static structure.

2.2 Static Structure

The static structure of the RUP provides a detailed description of the overall process. Four modeling elements compose it (Krutchen, 1999):

- Workers
- Activities
- Artifacts

- Workflows

The worker elements define roles played by developers. While a many-to-many relationship may exist between workers and developers, our model defines a one-to-one mapping between the two sets for simplification purposes. The Project Manager and Designer workers, for example, feature prominently in the model. Activities address the question of "how?" in an RUP project. These are work units that can be addressed to individuals acting as a particular worker. For example, *Find use cases and actors* is an activity assigned to the system analyst worker. Activities are represented metaphorically in this research. According to Leffingwell and Widrig (2000), an artifact is a piece of information controlled, modified, or created by a worker. As work products, artifacts are "what" the RUP produces in the form of tangible outputs. Most artifacts are considered part of a configuration and are therefore subject to change and version control. While RUP artifacts cover a wide range of activities, the artifacts modeled in this study concern analysis, design, and implementation. Krutchen (1999) defines a workflow as a succession of activities producing an outcome of discernable value. Workflows answer the question of "when" to do something in RUP. The Design and Implementation workflows are of particular interest in this simulation study. It is important to realize that workflows occur in parallel rather than sequentially. In fact, they run orthogonal to the time-sequenced elements of RUP, which comprise the dynamic structure of the process (Krutchen, 1999).

2.3 Dynamic Structure

A lifecycle is to a process what a melody is to a song. On a quantitative level, it controls how a process unfolds. From a qualitative perspective, it is the most recognizable part of

the process. Both these observations are true in the case of RUP. As shown in Figure 2.2, it is an iterative, incremental process superimposed on a modified spiral process. We consider each of these aspects individually.

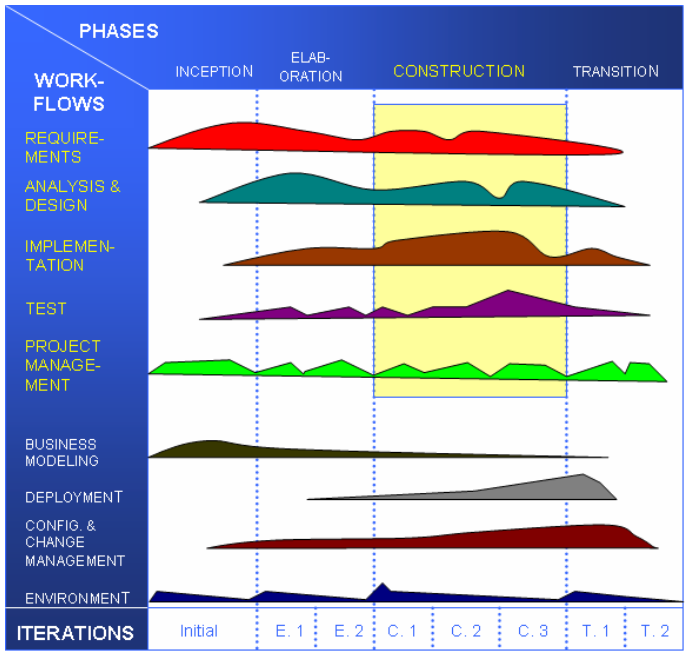

Figure 2.2. The RUP lifecycle (adapted from Larman, 2004; Kruchten, 2003)

Iteration allows RUP to cope with two major causes of project failure: change and unforeseen consequences. Change is particularly prevalent in the form of requirements churn. The customer's needs change because of a development in the market. The customer is dissatisfied with a partial product and corrects an earlier miscommunication or lack of communication. New government regulations imposed on the customer's industry could also cause requirement churn (Fowler, 2004). Other forms of change, however, often arise during a project. Turnover within the development organization is the most obvious example. Also, new tools may become available to the organization. In the software industry, geographic redistribution of personnel is a common source of

change (Pollice, 2004). The existence of unforeseen consequences is closely tied to change.

Human beings can rarely predict every outcome of their actions. Since humans create it, software encodes this uncertainty. Despite thorough planning, problems still arise. Components may be inconsistent or incompatible. The software may not account for certain hardware limitations such as restricted memory. These technical surprises as well as the previously discussed change inevitably lead to a certain amount of rework (Krutchen, 1999).

Iteration minimizes the amount of rework that has to be performed. Like all iterative processes, RUP repeatedly focuses on a subset of the requirements until all requirements are met or a deadline is reached. This technique allows the developer to delve deep into a project early in the process and discover potential pitfalls. It also gives the customer an early impression of the final product that will evolve under the current requirements set. This insight allows changes to be made near the beginning of the project minimizing wasted effort. Also, iteration maximizes staff utilization on a particular project. More employees can work on the same project at the same time. Two characteristics, however, separate RUP from other iterative processes (Krutchen, 1999).

RUP iterations are time-boxed and risky requirements are considered in early iterations. A time-boxed iteration is one with fixed start and end times. A certain number of activities are scheduled for each time-box. Rather than move a deadline, activities are moved to later iterations. Time-boxing shields an iteration's work product from becoming outdated. It also has a positive psychological impact on developers and customers who can regularly observe tangible results. By addressing risky requirements early in a

project, the development organization can avoid last minute crises, the resolution of which requires significant alteration to earlier work (Krutchen, 1999). This technique also provides an opportunity to abandon part or all of a project before substantial amounts of time and money have been wasted (Leffingwell and Widrig, 2000). Many of these benefits are enhanced by RUP's incremental nature.

The word "incremental" means different thing to different people. In RUP, "incremental" refers to the way the software product grows. In other processes, multiple new additions are merged with previous work products to create a new build. In RUP, additions are made one at a time. First, a small piece of software is coded and tested. Then, it is added to the working product, and the combination is tested. Finally, the tested combination is placed under configuration management, and the cycle repeats. The product is incremented within each iteration. The benefits of incremental development complement RUP's iterative nature (Krutchen, 1999).

Three main advantages arise from incremental development. First, fault location is simplified. Because the baselined portion of the new build has been more thoroughly tested, any faults are more likely to be found in the new addition. Also, components are exercised more thoroughly, thereby decreasing the chance of having latent defects. As was the case with iterations, incremental development provides early, tangible results. Having a running system boosts confidence in the project among customers and developers. Although iterations and increments serve valuable purposes, additional process elements are needed to evaluate progress on a project-wide basis. RUP services this need by overlaying the collection of iterations with a modified spiral process (Krutchen, 1999).

Since iterations have a depth-first "feel" to them, it is easy to get the impression that they are all the same and their order of completion is irrelevant. While it is closer to the truth on a small project, this misperception is grossly inaccurate on a large project. An early iteration might focus more on requirements and architecture, whereas an iteration late in the project might focus more on implementation and testing. To reflect this reality, RUP maps iterations onto phases. These phases correspond to the regions of a traditional spiral model (Krutchen, 1999).

RUP divides a (spiral) cycle into four phases: Inception, Elaboration, Construction, and Transition. Although their composition varies with each cycle, each phase has some general characteristics. In Inception, the scope and vision of the project are defined. The organization may also specify a business case during this phase. Elaboration is characterized by the development of an architecture, the planning of activities, and the gathering of resources. During Construction, work products from earlier phases are evolved and the product is actually built. In the Transition phase, the product is delivered to the user and maintenance activities are performed. At the end of each phase, a milestone is reached that serves as a point to evaluate the overall progress of the project and to affirm the practicality in continuing. The exact nature of the phases will also depend on the context in which RUP is applied (Krutchen, 1999).

CHAPTER 3

Conceptualization of Team-RUP: Organization, Task, and Performance Models

Dispelling the myth that software is manufactured was not a recent development. The production of software has been recognized as a creative process for quite some time (Pressman, 2005). Unfortunately, the study of software development often mirrors that of manufacturing processes and fails to account for the implications of software's uniqueness. The foremost among these implications centers on the creators' relationships. Due to mechanization, manufacturing teams are often best characterized as secondary social groups. Communication is often standardized, formal, and minimized. In contrast, software teams are primary groups. Even when roles are specialized, the creative process necessitates open communication. (Tischler, 2002).

The Team-RUP model explicitly addresses collaboration and coordination of agent team members via communication. In fact, the representation and measurement of aspects related to these concepts pervade and unify the three components of the Team-RUP framework. To decrease coupling and allow for future modification, we partition the Team-RUP conceptual model into disjoint sub-models: the organizational model, the task model, and the performance model (Figure 3.1). Collectively, these models describe the structure of the virtual organization, the work it performs, and how well it performs the work.

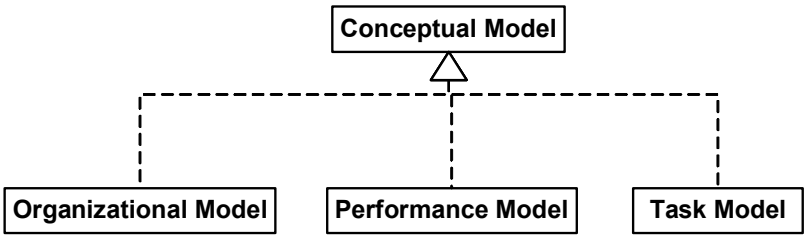

Figure 3.1. Realization of the Conceptual Model

1. Organizational Model

The organizational model addresses the structure of the organization and agents, the coordination of tasks, and agent collaborations. As is most common among businesses, the Team-RUP organization is structured as a hierarchy of agents. It consists of a project manager, a design manager, and teams of engineers. The project manager performs high-level coordination tasks, whereas the design manager provides oversight to the design process and decomposes tasks of large granularity. The teams of engineers create the actual artifacts of the software development process. The remainder of the software development organization, including an independent testing group, is implemented using standard object orientation. The structure of an organization modeled in the Team-RUP framework is shown in Figure 3.2.

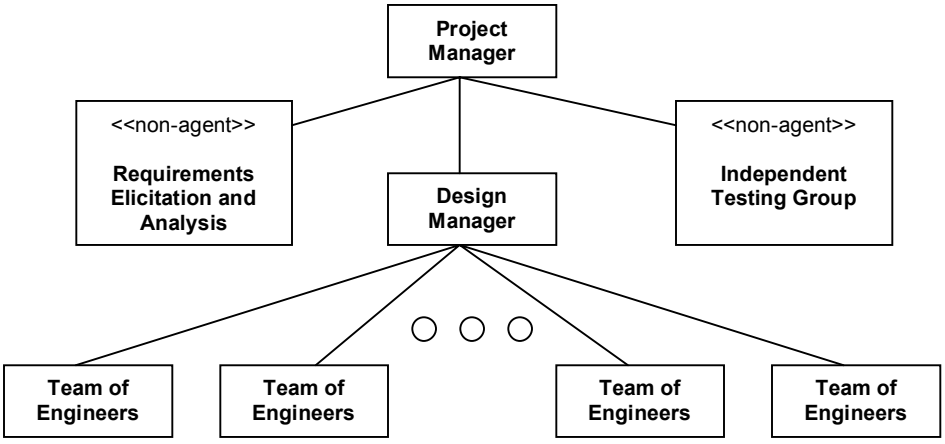

Figure 3.2. The Structure of an Organization in the Team-RUP Framework

To represent cooperation at the team level, this model considers four group archetypes based on characteristics resulting from collaboration and coordination techniques. Ferber (1999, p. 80) defines a collaboration technique as "being of those that enable agents to distribute tasks, information and resources (among themselves) in the advancement of a common labour." We classify teams in terms of the degree of autonomy afforded by such strategies. In particular, team collaboration strategies are classified as top-down or bottom-up. As the former entails step-wise refinement, a large degree of oversight is required, which diminishes autonomy. The latter, however, provides much more flexibility since the structure of the final integrated product is not entirely preconceived. These categories are further subdivided in terms of coordination. According to Ferber (1999, p. 400), coordination of actions means "the articulation of the individual actions accomplished by each of the agents in such as way that the whole ends up being a coherent and high-performance operation." We classify team behavior according to the degree of concurrency realized through coordination. In particular, teams can function sequentially or concurrently. Varying degrees of collaboration and concurrency, therefore, lead to four team archetypes. These archetypes match closely with Constantine's (1993) four organizational paradigms. The four team archetypes in Team-RUP are Synchronized, Concurrent, Agile, and Autonomous.

1.1 Synchronized Teams

Synchronized teams approach problems in a linear, top-down fashion. Such teams work under a traditional hierarchy and have specialized skill sets. At each stage of step-wise refinement, task dependencies are delineated and work progresses along the dependency

chain in a sequential manner. Synchronized teams correspond to Synchronous teams in Constantine's (1993) framework.

1.2 Concurrent Teams

Concurrent teams approach problems in a concurrent, top-down manner. Like Synchronized teams, they have a hierarchical structure. Task decomposition, however, creates a less flexible framework that is based on a centralized design strategy. This initial coordination effort minimizes the time and effort that must be expended on system integration activities. An essential characteristic of Concurrent teams is that decisions are handed down from above. Concurrent teams correspond to Closed teams in Constantine's (1993) framework.

1.3 Agile Teams

Agile teams approach problems in a linear, bottom-up fashion. Collectively, these teams have some idea concerning the direction in which a project needs to go. They work together to determine who does what in a particular situation. Agile teams complete their tasks in an incremental manner, adding new features until a final product is achieved. The label "Agile" derives from the fact that many agile processes such as Extreme Programming and DSDM advocate this form of team behavior (Pressman, 2005). Agile teams are similar to Constantine's (1993) Random teams.

1.4 Autonomous Teams

Autonomous teams follow a concurrent, bottom-up approach. Teams work independently of one another to create pieces of a solution. These pieces must be integrated to form the final system. A higher value is placed on innovation than on organization. Therefore, a large amount of effort must be expended to patch the system

together. Constantine's (1993) Open teams fit well with Autonomous teams. Figure 2 summarizes team behavior in Team-RUP.

Team Archetypes		Degree of Autonomy in Collaboration	
		Top-Down	Bottom-Up
Degree of Concurrency in Coordination	Concurrent	Concurrent (Closed)	Autonomous (Open)
	Linear	Synchronized (Synchronous)	Agile (Random)

Figure 3.3: Team Behaviors

1.5 Emergent Behavior

The final facet of the organizational model concerns emergent behavior. Each agent has a certain level of experience, which increases as simulation time passes. At the start of a simulation run, the experience of every agent is randomly assigned. Greater experience decreases the likelihood of that agent making a mistake. How an agent makes a mistake is determined by the type of agent as well as the context of the situation.

2. Task Model

In accord with RUP, the development of software in Team-RUP is viewed as a multi-stage transformation. Representatives of the development organization elicit requirements from the customer and users. These requirements are validated, refined, and translated into an analysis model. Team-RUP models perform these tasks procedurally. The agent organization becomes involved when software construction takes place.

Construction begins with the analysis model being passed to the project manager who forwards it to the design manager. During a series of time-boxed iterations, the design manager and its subordinates map the analysis model into a design model and implementation. For simplification purposes, the latter two artifacts are not syntactically distinct entities in the model. As will be evident later, both are sub-lists of the same (semi-) sorted list. To reflect RUP's incremental nature, testing occurs during each iteration.

We recall that in the Rational Unified Process, software projects are completed through a sequence of time-boxed iterations, each of which may comprise design, implementation, and/or testing (Krutchen, 1999). Development organizations using RUP should complete some subset of the final system at the end of each iteration. Because of its iterative nature, however, RUP allows an organization to cope with requirements changes. RUP also stipulates that those parts of the project involving greater risk should be addressed in early iterations so that overall risk to the project can be mitigated. To model this incremental and iterative process, the organization sorts integer arrays using a

variation of a well-known iterative sorting algorithm; namely, Shell sort (Krutchen, 1999, Pollice, 2004).

2.1 Project Configuration

In the Team-RUP framework, an array of integers represents a project configuration, and problem facets are modeled as ordered pairs of elements. The set of all possible facets pertaining to a configuration C (and hence a project) is the following set:

$$\{(x_i, y_j) \in C \times C | i < j\}$$

Suppose (x_i , y_j) is a problem facet corresponding to a configuration C and let z_k be an element of C. If $k \geq j$, comparing x_i and z_k translates into performing a task associated with (x_i , y_j) . Similarly if $k \leq i$, comparing z_k and y_j is also analogous to performing a task associated with (x_i , y_j). Clearly, many tasks can be accomplished via a single comparison. As in the real world, not all facets of a problem need to be addressed to complete a given project; certain tasks can remain undone. Team-RUP classifies tasks according to two categories. A supporting task does not reverse the order of a problem facet pair. Principal tasks are the second form of task, and they address a special form of problem facet.

Team-RUP represents requirements in terms of inversions; that is, pairs of array elements that are out of order. A single inversion is interpreted as a principal, *atomic* task yet to be performed. Thus, a set of inversions is an incomplete requirement fulfillment (i.e., principal task), and removing a set of inversions corresponds to fulfilling a requirement. Sets of inversions can be decomposed into subsets just as a task can be decomposed into subtasks. Each principal, atomic task is associated with a risk.

2.2 Risk Management

Mark Akhed (2003) defines risk as "an ongoing or upcoming concern that has a significant probability of adversely affecting the completion of major milestones and product quality". That is, a risk is something that (if it happens) prevents requirements from being fulfilled. Although a broader definition could be adopted (in which this sort of risk would be classified as a *technical* risk), a requirements-centered meaning is most relevant to the simulation of a generic software company. Note that a risk has not occurred. When a risk develops into actuality, it ceases to be a risk and becomes an *issue* (Akhed, 2003). Therefore, risk projection is an essential part of managing risk. Risk projection involves ranking risks by their likelihood of occurrence and their potential impact (Pressman, 2005). Thus, a less costly issue that has a high probability of occuring may be more risky than a more costly issue with a smaller probability of occuring. Observe that risk varies throughout a project. RUP recommends that requirements involving greater risk should be performed in early iterations (Akhed, 2003).

The Team-RUP model captures the concept of risk by extending our earlier analogy involving inversions. Because risk estimation precedes the development of a risk mitigation strategy, we initially assume that sorting takes place through the iterative swapping of randomly chosen array elements. The risk probability R_{ij} associated with a particular inversion i at the instant before the j^{th} swap of the project should be defined in a manner adhering to the analogy of inversion removal as requirements fulfillment.

Ultimately, we must associate R_{ij} with the probability that i will not be removed during the course of the project. Assuming a *success* occurs when swap j removes inversion i, define the random variable X_{ij} to be 1 in case of a success and 0 in case of a

failure. Since the probability that X_{ij} records an inversion removal can be determined entirely from the state of the requirements array, it follows that the sequence $\{X_{ij}\}$ is a Markov Chain for fixed i and varying j. Note, however, that the X_{ij}s are dependent. It might appear that risk probability for a project involving n swaps should be defined as follows:

$$R_{ij} = P(X_{i,j+1} = 0, X_{i,j+2} = 0, \ldots, X_{i,n-1} = 0, X_{i,n} = 0)$$

This definition, however, fails to draw a distinction between actual and perceived risk probability.

In the real world, risk probabilities are typically derived using the Delphi technique, which essentially involves the iterative polling of experts (Charette, 1989); that is, via opinion. Such opinion is founded upon the forseable nature of the requirement. It would be erroneous to model risk probability as an actual probability based on the entire remainder of the project, during which the perceived nature of the requirement could change. After all, the risk associated with requirements change (i.e., project risk) is typically considered separately from technical risk (Pressman, 2005). Instead, the calculation of the risk probability of a technical risk should assume the requirement stays the same for the rest of the project. This assertion makes clear the need for frequent, iterative risk assessment.

In Team-RUP, the risk probability associated with an inversion at a particular point in time is defined as the probability that the inversion will be removed before the end of the project with the assumption that neither of the elements of the inversion change array locations before the inversion is removed. This assumption makes the X_{ij} independent of one another. It is also assumed that the inversion will never be

reintroduced. Therefore the probability of removing the inversion via the next swap does not change from swap to swap. In other words, we have reduced the Markovian chain problem to a simple binomial experiment. The risk probability associated with the inversion is the probability that every trial will result in a failure. Since a numerical value for risk probability is not calculated in the current implementation of Team-RUP, the important observation to make centers on the relative probabilities of different types of inversions.

In particular, we observe a difference between inversions involving array locations that are far apart and those involving array locations that are close together. To remove an inversion, at least one of the inversion elements must be chosen for swapping. Furthermore, the other array element chosen for swapping cannot lie between the two inversion elements. For example, there is no swap involving 3 that will remove the inversion (4,2) from the list 5, 4, 3, 2, 1. Therefore, inversions consisting of elements that are close together in terms of array location have a higher probability of being removed. Stated another way, inversions consisting of far-apart elements have a greater risk probability.

2.3 Shell Sort

With the mapping of tasks to inversions, Shell sort provides a strong analogy for RUP for several reasons. Obviously, the algorithm's approach of removing inversions in phases coincides with RUP iterations. Of greater significance is the fact that a Shell sort phase does not undo work performed in earlier phases: 2-sorting a 5-sorted list generates a 5-sorted (as well as 2-sorted) list. This reflects the fact that each RUP iteration produces

part of the final system rather than draft-quality, throw-away workproducts. Shell sort also captures RUP's risk mitigation strategy.

Recall that Shell sort initially swaps unordered elements that are far apart and decreases with each phase the distance between the elements it compares. Because it eventually sorts the set of adjacent elements, the algorithm is guaranteed to sort the entire list. We see from Section 2.2 that Shell sort addresses requirements with high risk probabilities during early iterations. Given two requirements with the same risk probabilty, does Shell short differentiate between the requirements based on risk cost? The answer is yes, but the reasoning is subtle.

Shell sort can be thought of as a tournament (not to be confused with the tournament sort algorithm). Each phase of Shell sort is a "round". "Matches" occur between elements that are kn array locations apart, where k is an integer and n is the Shell increment. If we think of higher numbers as better players, we can make the following observation: With each round of the tournament, better numbers move to the end of the array, mediocre numbers move to the middle, and less skilled numbers move to the front of the array. The further the tournament progresses, the more defined this ordering becomes until finally the number are completely sorted. If the tournament is cut short, however, a general relative trend will still exist: small numbers will appear near the front of the array, medium numbers will appear near the middle of the array, and large numbers will appear near the end of the array. At such a point, two numbers which have a large difference in terms of absolute value are more likely to appear in sorted order than two that differ little in value. In other words, inversions representing a higher risk cost are removed during earlier iterations. Therefore Shell sort captures RUP's suggestion

that requirements involving high technical risk be addressed in early iterations. As mentioned in Section 2.2, however, changes in requirements involve another type of risk.

As discussed in Section 2.3, RUP assumes the presense of change. The iterative nature of RUP mitigates the negative impacts of this unavoidable reality. Because it represents a significant divergence from older processes, our analogy must somehow incorporate this characteristic. The key insight is that Shell sort will still properly order an array even if none but the last sorting phase actually occurs. Since inversions inserted by an outside entity during the execution of Shell sort will ultimately be removed by later phases, the algorithm reflects RUP's ability to cope with changing customer requirements.

As mentioned before, we interpret Shell sort phases to be RUP iterations. In each phase of Shell sort, the base array is partitioned into sub-arrays whose lengths are determined by a particular increment sequence. Each of these sub-arrays is sorted using some secondary sorting algorithm (typically insertion sort) (Weiss, 1999). By varying the way this secondary sorting occurs, we use this feature of Shell sort to model alternative team behaviors. Comparing array elements and performing element exchanges represent the design and implementation phases. Ascertaining the number of inversions in sub-arrays is seen as testing. Several sub-arrays can be tested during each iteration. Thus, in accord with RUP, our model organization avoids a linear lifecycle approach (Pollice, 2004). The project ends when the originally proposed deadline for the final time-box expires, regardless of the state of the array.

3. Performance Model

To maximize the applicability of the results across the vast diversity of software development objectives and constraints, efficiency and effectiveness have been chosen as the principle indicators of utility in this study.

3.1 Software Metrics

We measure efficiency in terms of productivity and staff utilization. Effectiveness is viewed as a combination of timeliness and software quality. In accord with Practical Software Measurement (Office of the Under Secretary of Defense for Acquisition and Technology [OUSDAT], 1998), productivity P is defined as the ratio of the product size to the effort, where effort is typically quantified in terms of labor hours. Taking a functional view of product size, we define this quantity as the total number I of inversions removed from the requirements list. Effort E is the product of the number of agents A and the number of time ticks T in the simulation. That is,

$$P = \frac{I}{TA}$$

The formula for staff utilization Su also contains effort in its denominator. The numerator, however, equals the total number Pt of "productive ticks" taken across all agents. A time tick is considered productive *for a particular agent* provided that agent performs some work-related activity during that time tick. For example, consider a three tick experiment with two agents. If one agent works during every tick and the other agent only works during the second tick, we have $Pt = 4$. In summary,

$$Su = \frac{Pt}{TA}$$

Timeliness is measured as a function of build content. The build content measure is the number of components *Cc* completed during an iteration divided by the number of components *Ca* allotted to the iteration at its start (OUSDAT, 1998). In this study, the granularity of a measurable component is the same as that of a testable component: a single cell of an *n*-cell partition of indices created during an attempt to *n*-sort the values corresponding to those indices. Timeliness *Tm* is the mean value of the build content measures taken over all iterations. In an experiment with *n* iterations, we have

$$Tm = \frac{1}{n}\sum_{i=1}^{n}\frac{Cc_i}{Ca_i}$$

Quality *Q* is measured in terms of defect density. PSM defines defect density as the number of defects in a component divided by the size of that component (OUSDAT, 1998). Translating this definition into the language of the Team-RUP model, quality equals 1 minus the number of remaining inversions *Ri* divided by the total number *In* of inversions in the original requirements list:

$$Q = 1 - \frac{Ri}{In}$$

CHAPTER 4

The Design and Implementation of Team-RUP

Much of the design model follows directly from the problem statement as outlined in Chapter 1. For example, the mapping between RUP iterations and Shell sort phases is immediate; sorting algorithms in general match well with process lifecycles. Mapping team behaviors to sorting algorithms, however, is far more complicated.

1. High Level Overview

We first take a look at how the software organization as a whole operates. That is, we can consider the workflow of a simulated software development project. A software project is not the simple execution of a predefined sequence of atomic steps. Rather, it is a progression through a series of interdependent states governed by a complex system of time-dependent conditions. To illustrate this workflow, we utilize a UML activity diagram.

Figure 4.1 reveals the workflow of Autonomous teams during a single iteration. Although it focuses on a single team type, it illustrates high-level coordination activities common to all the behaviors. The project manager is ultimately responsible for starting and stopping each iteration. At the start of an iteration, it tells the design manager to implement a set of components. The design manager chooses one of the components and creates a work breakdown structure. Each leaf of this tree represents a sub-component to

be implemented by a single team. These sub-components are assigned to teams until either the leaves or the available teams are exhausted. As teams finish sub-components, the assignment process continues. Once some of the leaves have been addressed, the design manager may assign interior nodes.

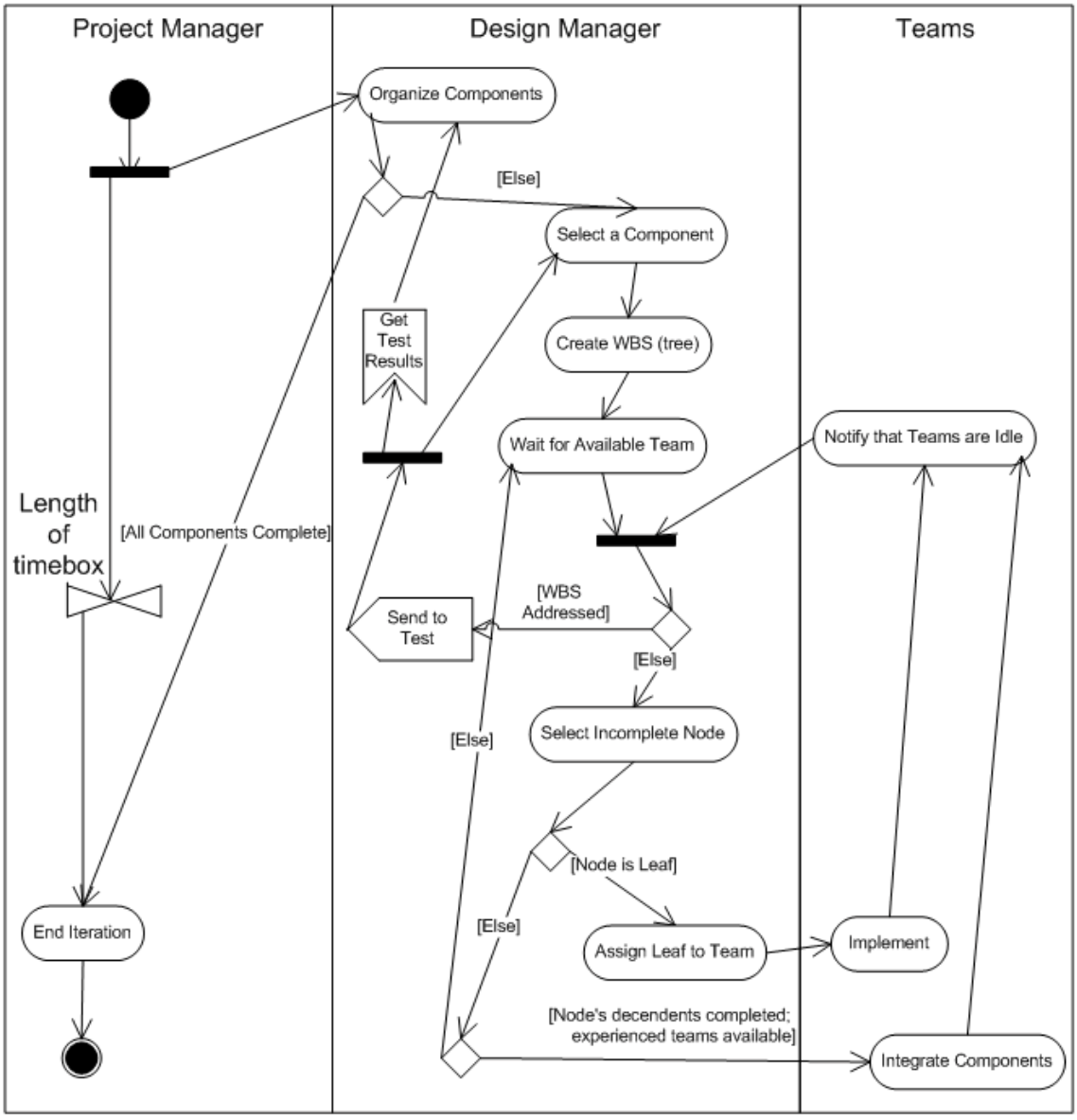

Figure 4.1. Iteration Workflow for Autonomous Teams

Interior nodes can be assigned to teams once all their descendents have been completed. Interior nodes represent integration tasks and are assigned to two teams. Each team should have previously been assigned a child of this node. They collaborate to integrate the sub-components. Once all the nodes of a work breakdown structure are addressed, the completed component is sent to the testing division. If the testing division does not find it acceptable, the component is returned to the design manager's queue of components to implement during the current iteration. The construction division continues the process of designing and implementing components until the component set is exhausted or the time-box expires.

The workflow presentation presented in the preceding paragraphs is specific to autonomous teams only in the handling of each component. For example, concurrent teams perform integration tasks throughout component implementation rather than delaying the task until (potentially) all sub-components have been completed. The basic steps of starting an iteration, implementing a set of components, testing each component, revisiting components that fail to withstand testing, and ending an iteration are common to each organizational archetype.

2. Team Types via Sorting Algorithms

A direct mapping between team behaviors and sorting algorithms would reveal nothing more than is covered in an algorithms course and would fail to capture member collaboration. The algorithm's *approach* to sorting, not its efficiency or exact details, is what needs to be captured. One successful match is made between Autonomous teams and merge sort.

2.1 Autonomous Teams via Merge Sort

Under every team behavior, each iteration begins with the design manager selecting components to be completed during that iteration. The scope of a team behavior's influence is a single component; i.e., a set of indices. As shown in Figure 4.2, when an Autonomous design manager considers a component, it first creates a work breakdown structure.

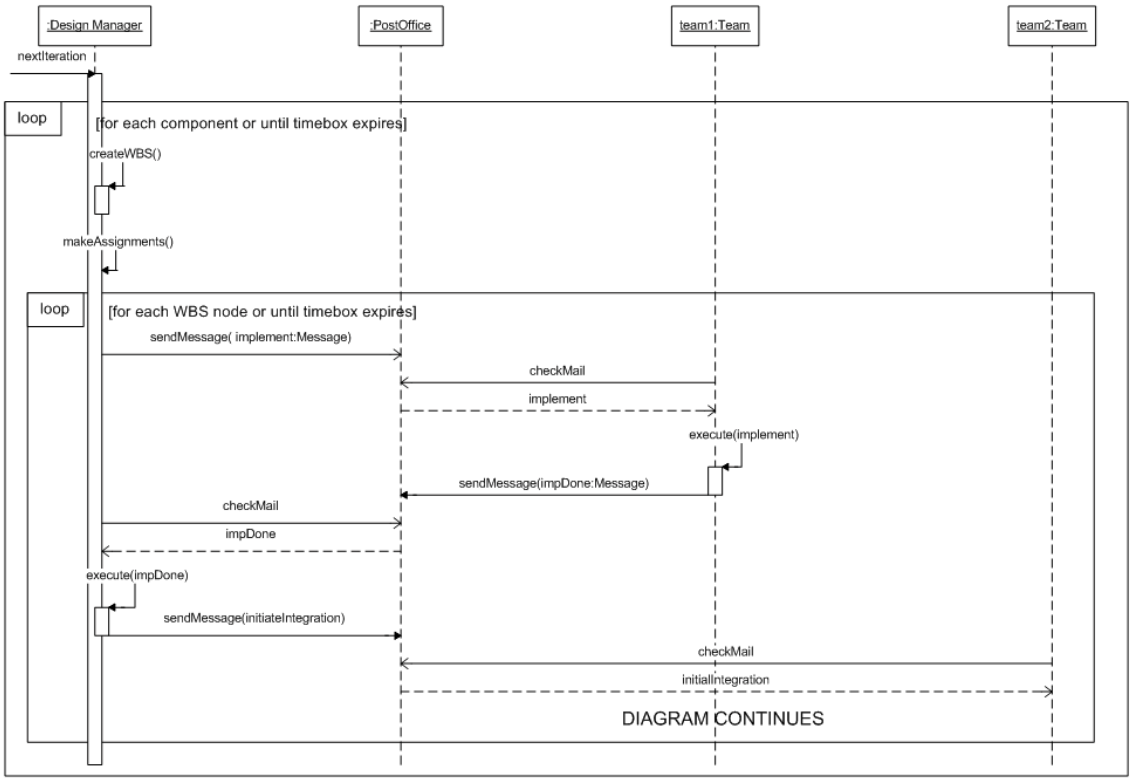

Figure 4.2. Operation of Autonomous Teams

This structure is a binary tree in which the root contains the entire component. Sibling nodes split the indices of the parent nodes. Each leaf node contains a set of indices no larger than the value of the workload parameter and has a parent that contains a set of

indices of size greater than the workload. The work breakdown structure corresponds to the tree of recursive calls in the standard version of merge sort.

Once the tree has been constructed, each leaf is assigned to a single engineering team, which sorts the sub-array determined by the node's index set. Moving from bottom to top in the tree, the design manager assigns each interior node to two teams as shown in Figure 4.3. These teams sort the sub-array determined by the node's index set by merging the sub-arrays in the node's two children. Whenever possible, the team assigned a parent node will have been involved in sorting one of the child nodes. The two teams assigned to a node each manage a single child node and must collaborate to sort the sub-array determined by the union of the sets of indices contained in the child nodes. The way the two teams collaborate to merge the sub-arrays is very similar to the method used in the standard merge sort operation. A similar mapping exists between Concurrent teams and the quick sort algorithm.

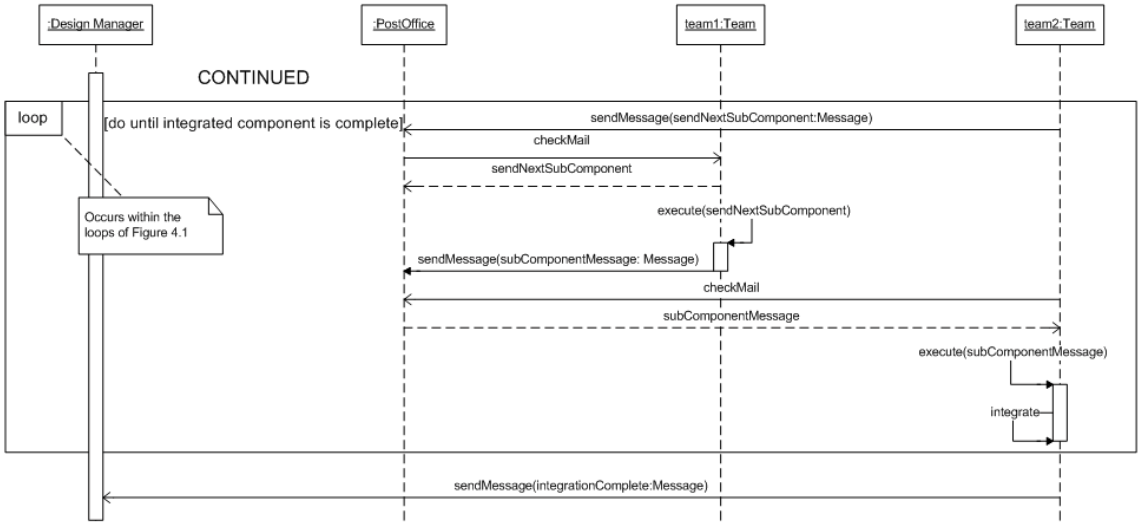

Figure 4.3. Operation of Autonomous Teams (continued)

2.2 Concurrent Teams via Quick Sort

The implementation of quick sort in the Team-RUP model closely resembles that typically found in a textbook on algorithms. As shown in Figure 4.4, when the design manager considers a component, it enters a design phase that corresponds to the recursive partitioning in quick sort.

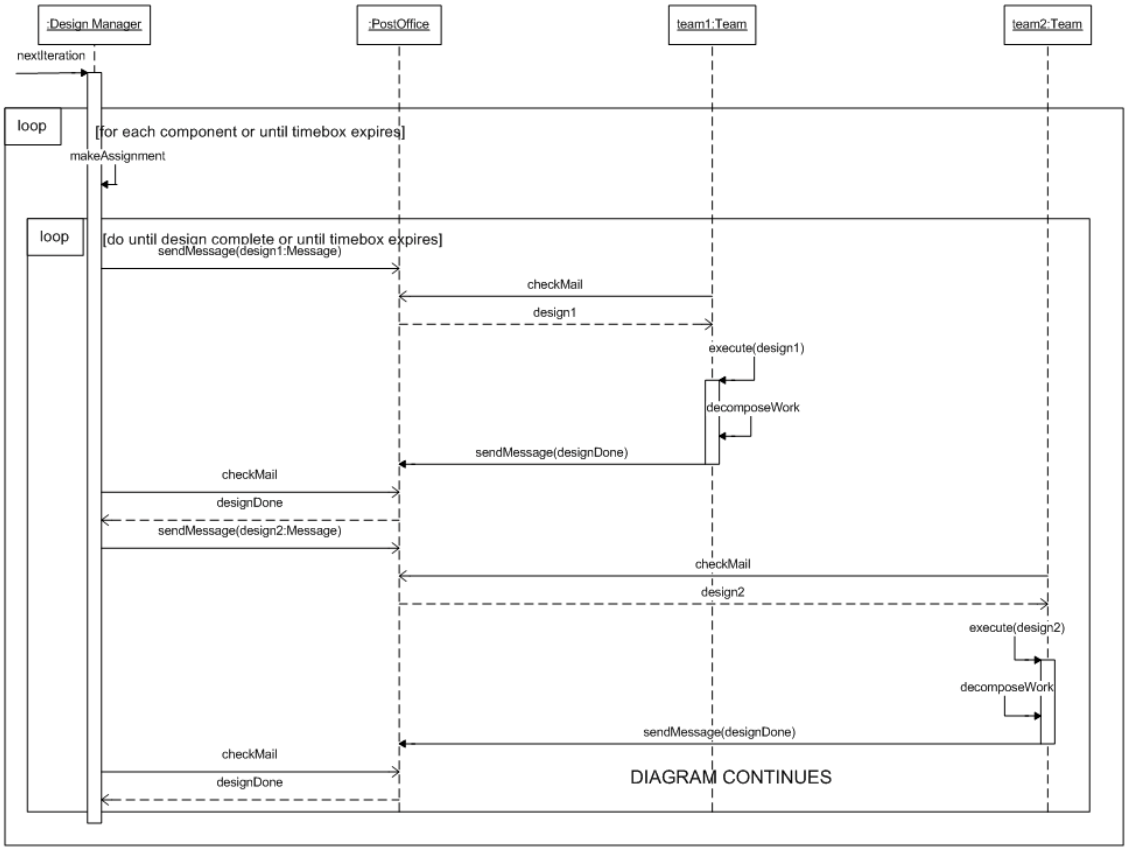

Figure 4.4. Operation of Concurrent Teams

A single engineering team partitions the array determined by the component. Each half of the partitioned array is itself partitioned by other teams. This process continues until the size of a sub-array is at most the value of the workload parameter. At this point, a team implements the sub-component; i.e., sorts the corresponding sub-array (Figure 4.5). The design manager is in charge of assigning sub-components to

engineering teams. The manifestation of insertion sort in Team-RUP is somewhat more complicated.

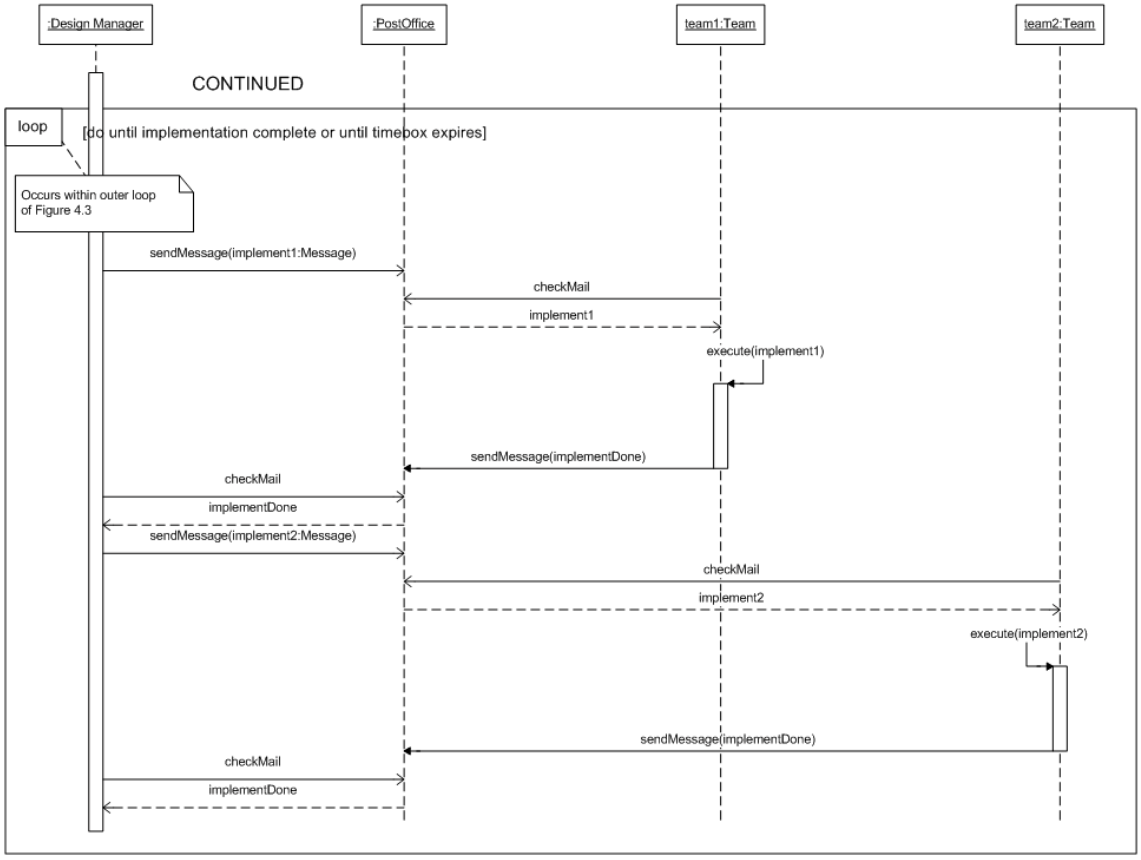

Figure 4.5. Operation of Concurrent Teams (continued)

2.3 Agile Teams via Insertion Sort

Insertion sort provides an analogy for the Agile behavior. For teams operating under this behavior, tasks corresponding to a given requirement are associated with a single engineering team. In concrete terms, this statement means a team manages a particular set of values regardless of where they appear in the array. This strategy is markedly different from the index-tracking strategies used with the other behaviors. The design manager must follow which teams are managing what values and what values are located in which index locations.

As shown in Figures 4.6 and 4.7, when the design manager considers a component, it logically partitions the array indices into sorted locations (the first index) and unsorted locations (all other indices).

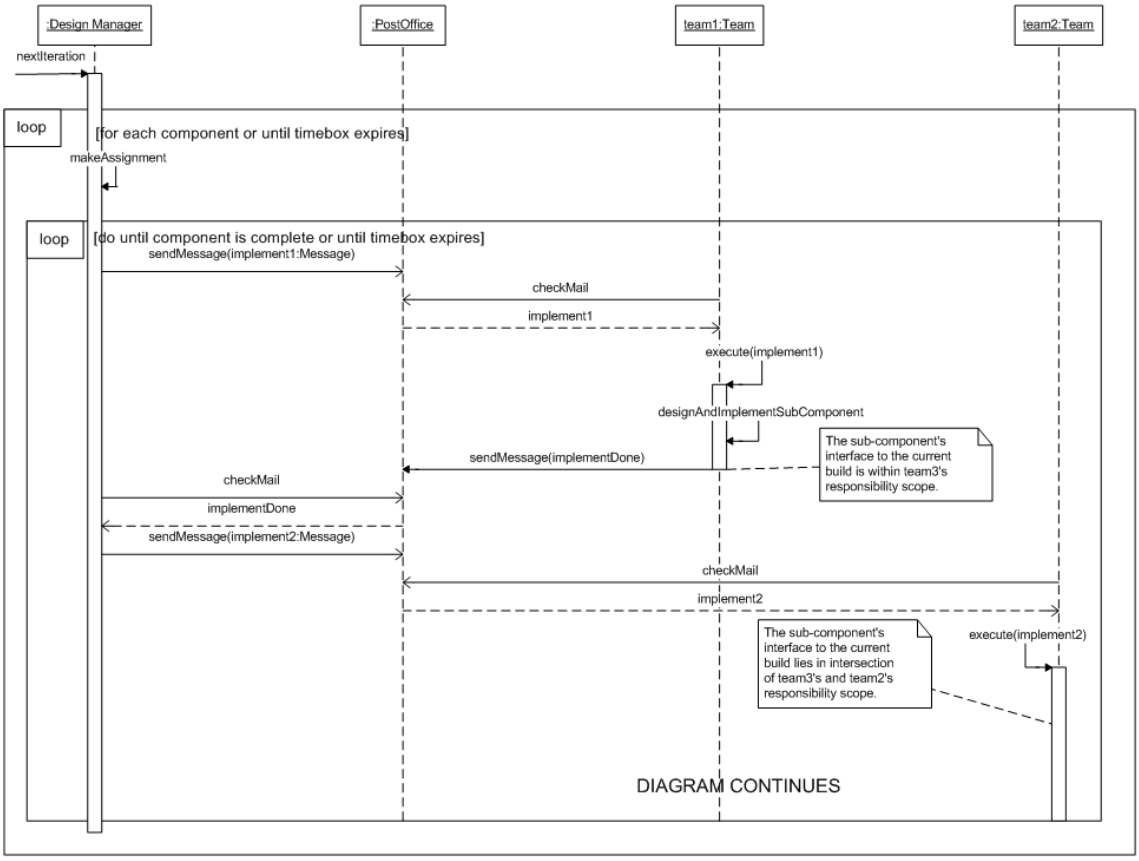

Figure 4.6. Operation of Agile Teams

In ascending order of indices, the design manager adds unsorted locations to the sorted locations list. First, the design manager contacts the team *T* in charge of the value in the next unsorted location. Team *T* contacts teams managing values in the sorted locations and exchanges *indices* with these teams until the value belonging to *T* is in the appropriate location. A value leaves the *active* state and enters the *passive* state once it has been inserted into the sorted locations. At this point, the inserting team can accept another value to be managed. At any point in time, the number of active values managed

by a team must not exceed the workload. The modified heap sort is the last and most complex algorithm modification.

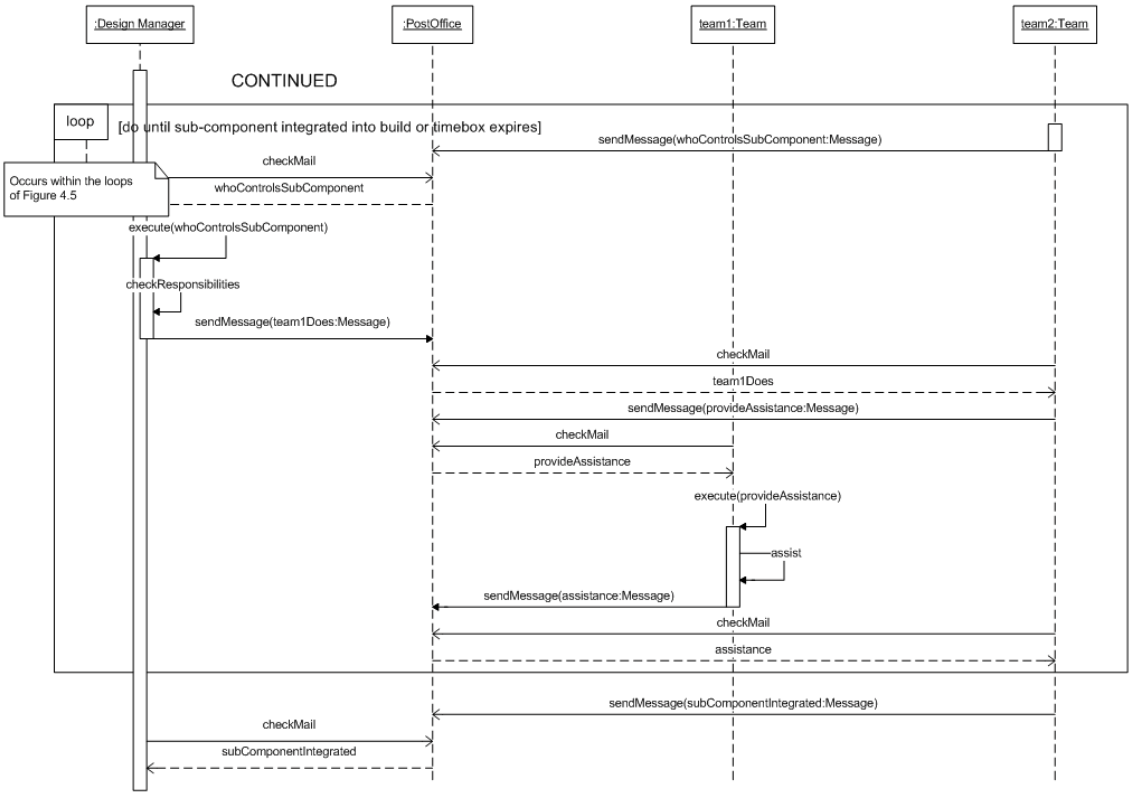

Figure 4.7. Operation of Agile Teams (continued)

2.4 Synchronized Teams via Heap Sort

Heap sort provides an analogy for the Synchronized behavior. As shown in Figures 4.8, 4.9, and 4.10, when a design manager first considers a component, the sorted array indices (not values) are divided into disjoint, contiguous sets. Each set represents an area of specialization and is managed by a single team. Only engineering teams that share a border communicate with each other. Collectively, the indices in the areas of specialization form a heap tree. The design manager gives a single, atomic order to heapify at a node or to delete the maximum element and reduce the size of the heap. The

teams collaborate to satisfy these demands passing values via the pipeline created by specialization. Blocking can sometimes occur in the pipeline.

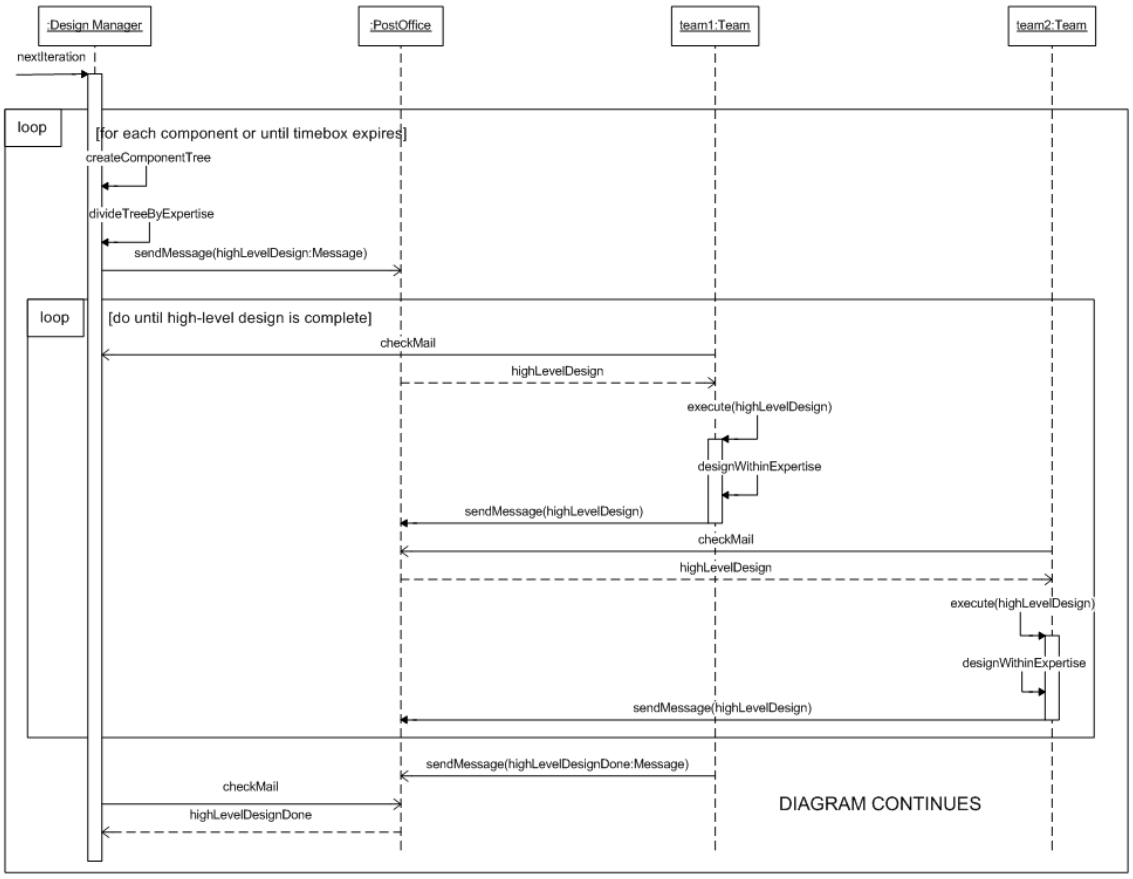

Figure 4.8. Operation of Synchronized Teams

Pipeline blockages are the result of special tasks associated with values when they enter a team's specialization boundary. A certain number of productivity units are attached to each of these tasks, and a value cannot cross a specialization boundary until the productivity units have been depleted by the specialist team. Each team can expend a fixed number of productivity units on its tasks during each time tick.

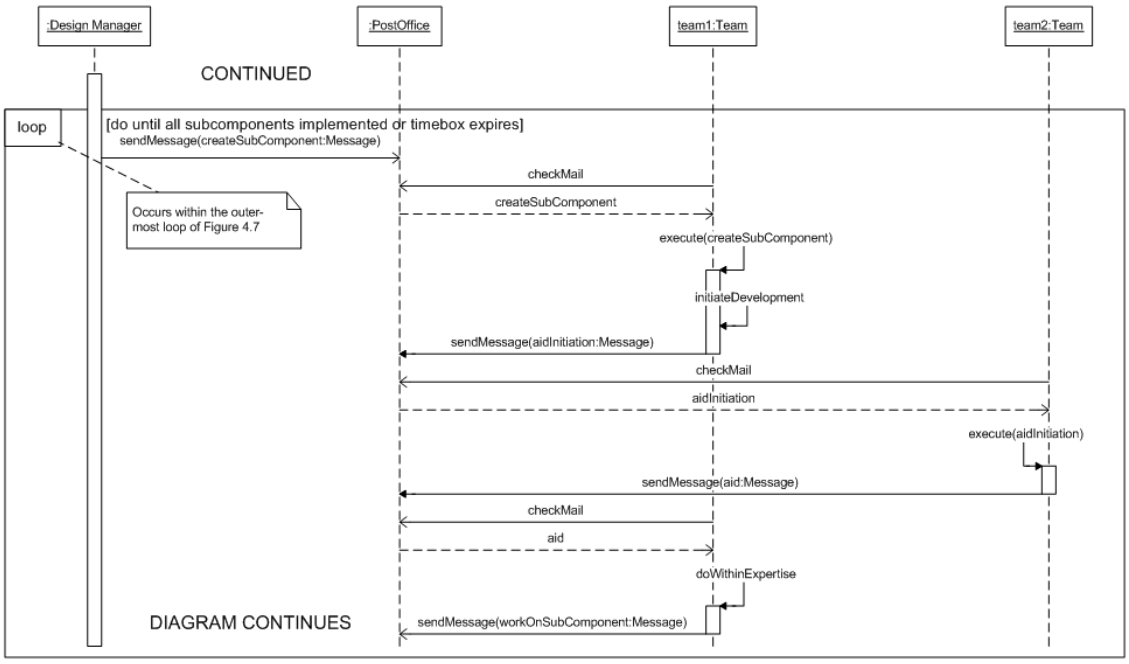

Figure 4.9. Operation of Synchronized Teams (continued)

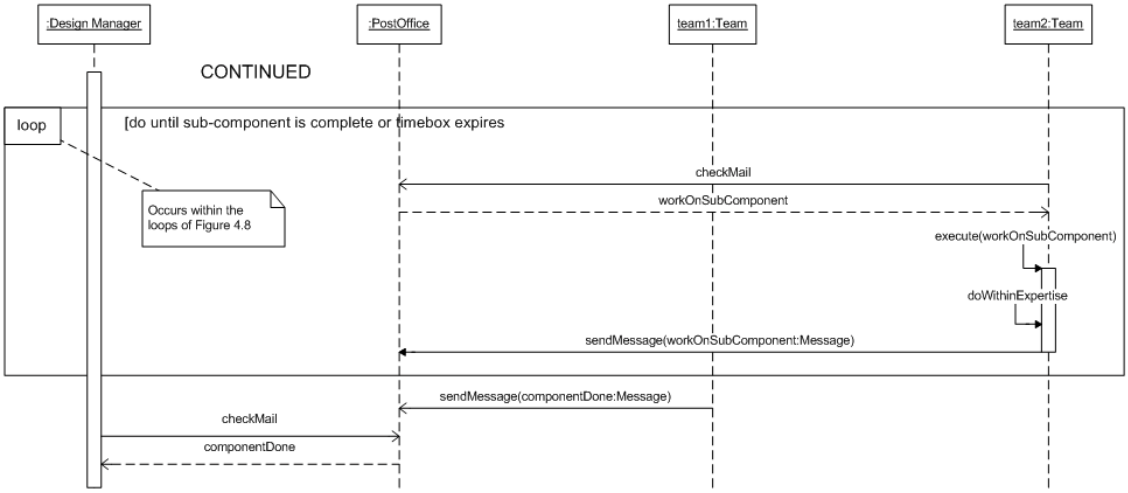

Figure 4.10. Operation of Synchronized Teams (continued)

2.5 Incorporating the Experience Factor into the Model

Experience allows otherwise identical agents to exhibit divergent behavior in identical scenarios. For example, the experience of a team is directly proportional to the

expediency with which it completes an implementation task. That is, an experienced team will consume fewer time ticks than an inexperienced team while implementing the same component. Each agent encodes its experience as number between 0 and 1. At the start of a simulation, experience values are chosen randomly. Whenever an iteration ends, each value is increased by a randomly-chosen, nonnegative increment that is at most half the difference of 1 and the current value. Experience updates are another example of scheduled events.

Experience also affects how well an agent performs its job. An experienced agent is less likely to "make mistakes" than a less experienced agent. In Team-RUP, making a mistake amounts to incorrectly comparing two array elements in terms of their size. Whenever an agent compares two numbers *a* and *b*, a random number is sampled from a uniform distribution. If this variate is less than the agent's experience, the agent correctly determines which of *a* and *b* is smaller. Otherwise, it performs the comparison incorrectly. Thus, team experience can indirectly affect each of the performance measures in Team-RUP. This inclusion of experience in the model promotes a tighter coupling with reality. While experience is an emergent behavior, it can be affected by turbulence parameters set by the experimenter. Because of its vast importance, we relegate a discussion of turbulence to its own chapter.

3. Design and Implementation

The structure of synthetic organizations in Team-RUP reflects that characterizing traditional software construction groups. In particular, agents are assembled hierarchically with a project manager agent at the apex. A design manager agent reports

to the project manager and has team aggregate agents as its subordinates. Each team agent encapsulates the roles of designer, programmer, and team leader. The number of team agents is a factor to be varied during simulation. Development activities falling outside software construction are realized as services performed by objects rather than agents. Agents communicate with hierarchical peers as well as direct superiors and subordinates. The package diagrams in Figures 4.11 and 4.12 reveal that the synthetic hierarchy is captured using a hierarchy of software components.

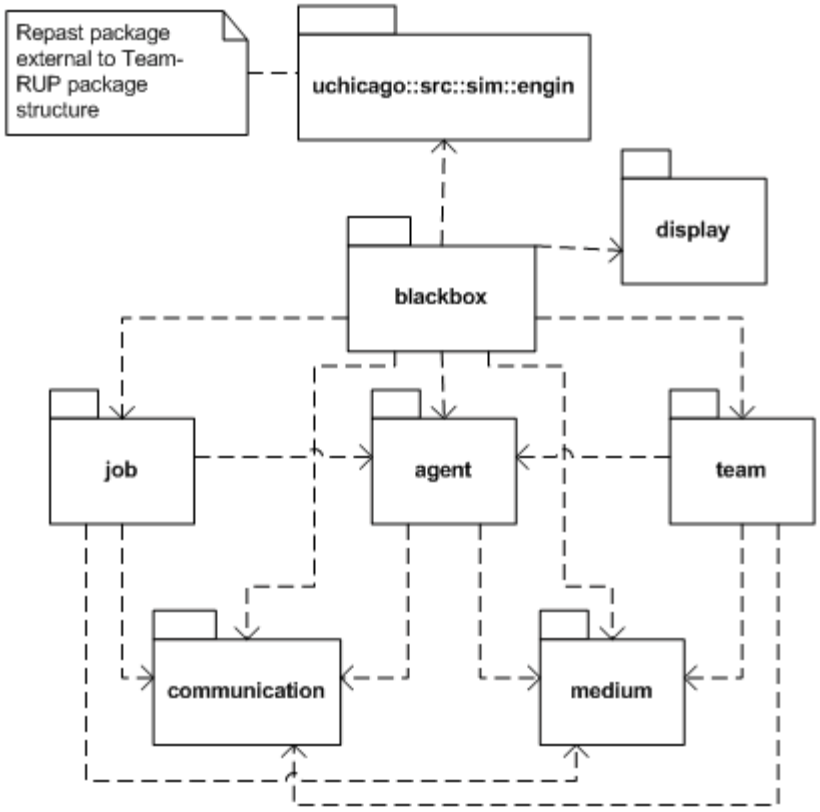

Figure 4.11. UML Package Diagram for the Overall Design

Communication is achieved via direct routing using an architecture patterned after postal service (Ferber, 1999). As shown in Figure A.1, each agent has a unique address, mailbox, and address book. To converse, an agent delivers to the appropriately addressed

mailbox, a message containing a performative, a return address, and any context-sensitive content. Agents periodically retrieve messages from their mailboxes and respond according to the environment and their current status. It is important to note that these addresses are not programmatic references or pointers. That is, agents are pairwise decoupled in the strictest sense. Though its use may reflect qualitative interpersonal characteristics in future implementations, the purpose of messaging at this point is strictly collaborative.

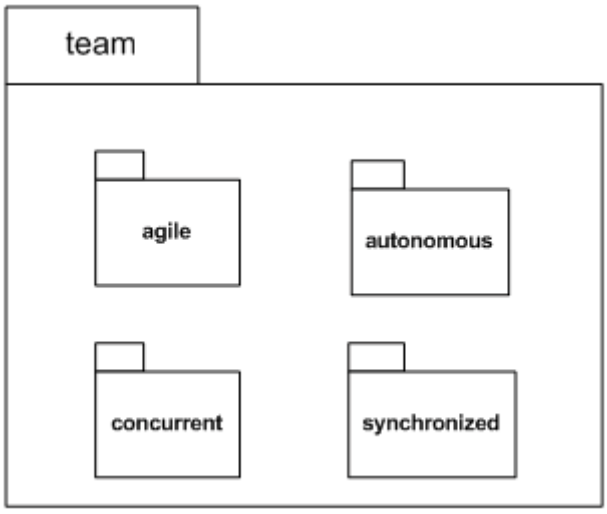

Figure 4.12. Content of edu.auburn.philljr.team package

In the Team-RUP model, the nature of collaboration varies with the selection of team behavior. Both centralized and distributed task allocation find representation (Ferber 1999). Task assignments involving the project manager are imposed. The design manager can exhibit trader qualities. For example, Agile teams consult the design manager to discover which team manages a particular requirement. In contrast, imposed allocation characterizes the autonomous team behaviors. Unlike teams of other types, Synchronized teams exhibit a form of distributed allocation. By utilizing acquaintance networks to ascertain the team whose skill set matches a service needed by the inquiring

team, Synchronized teams participate in direct allocation. Team interaction, another aspect of collaboration, exhibits even greater diversity among behaviors.

Each team behavior elicits a different interaction situation. Although an opposing argument could be made in cases of insufficient time, the teams in the Team-RUP model have compatible goals. Divergence arises in terms of resource and skill sufficiency. Synchronized teams, for instance, have unlimited access to the asset pool of the organization. Due to specialization among teams, however, interactions can be classified as simple collaboration. The symmetric situation marks Agile team interactions. Skill sets overlap, but a single team harbors the resources associated with any given requirement. Therefore, interactions among Agile teams exhibit obstruction. Independence, the simplest form of interaction, matches the Concurrent team behavior. Concurrent teams work cooperatively but without inter-team collaboration. Finally, Autonomous teams have neither sufficient skills nor resources. While seemingly paradoxical, this situation arises during component integration. A team can manage (with skill) only those components that it creates. If a component is paired with a team not involved with its creation, a time penalty is incurred. When two components are integrated, each team is limited to the resources associated with the component it previously created. Except in the case of Concurrent teams, the interdependence of team actions sometimes leads to conflict (Ferber, 1999).

From the enterprise perspective, teams serve the same goal of completing the development project. From the perspective of an individual team, however, the goal is completing whatever task has been assigned to it by the design manager as efficiently as possible. Due to skill or resource limitations, the efficiency qualification sometimes

leads to conflict. For instance, the case frequently arises among Synchronized teams in which a team must wait for one or more other teams to finish some activities before it can complete a task that has already been assigned by the project manager. This example illustrates one of the two conflict resolution strategies employed in Team-RUP. All things being equal, team requests are satisfied non-preemptively in a random order. Events are scheduled that recommence the activities of waiting teams when resources become available. The other conflict resolution technique is simple avoidance. The design manager orders tasks so as to minimize conflicts. For Autonomous teams, the design manager decomposes project units into a tree that partially orders tasks in terms of dependencies. Tree ancestors can only be completed after all descendents have been completed. Therefore, such an ancestor task is not even assigned to a team until the corresponding descendents are finished. The random allocation resolution technique resolves conflicts corresponding to unrelated nodes of the task tree (Ferber, 1999).

CHAPTER 5

Turbulence

Within a company, internal turbulence results from employee turnover. Employees can be fired or laid off. Also, they can quit their jobs. Internal turbulence significantly affects both the efficiency and effectiveness of an organization. Since these two characteristics are of principal interest in applications of the framework, Team-RUP explicitly addresses internal turbulence.

Three variables, each of which takes on a value between 0 and 1, influence the level of internal turbulence. The *base rate* parameter *B* represents the degree to which a company is downsizing. Its value has the greatest impact among these variables in determining the security and desirability of any given employee's job. With a high base rate, an employee is more likely to be fired or quit due to pressure or an ample severance package. Notice that Team-RUP treats firing and quitting as dependent events. While an individual's decision to quit his job may not be related to a projected payroll size, the rate at which such decisions are made for the company as a whole is closely linked in the general case. That we are concerned with a generic company is paramount. After all, a company that implements an unpopular decision (e.g., cutting employees' health insurance) may see an increase in the rate at which employees are quitting even if the company is not intentionally downsizing. The *sensitivity to performance* parameter *S* determines the degree to which an employee's performance affects his chance of

separating with the company. This number, of course, has little meaning without the *performance* variable *P*.

Performance is measured at the team level. Initially, the performance for each team is 0.5. It varies with respect to the team's current and overall historical productivity. In the Team-RUP framework, certain team behaviors, by definition, involve asynchronous development efforts. From a high-level, logical perspective, multiple teams change the locations of multiple elements in the requirement array at the same time. Since teams asynchronously manipulate interleaved sets of elements, attributing credit for particular tasks to individual teams presents a complication. It is not enough to know what team considered which locations. In Figure 5.1, for example, team A and team B asynchronously 2-sort an array consisting of 4 elements. From a logical perspective, the simulation progresses directly from step I to step III. Which team, therefore, deserves credit for completing task (4,3)? In general, we must introduce some rule for disambiguating such case, which could involve inversion removal or introduction.

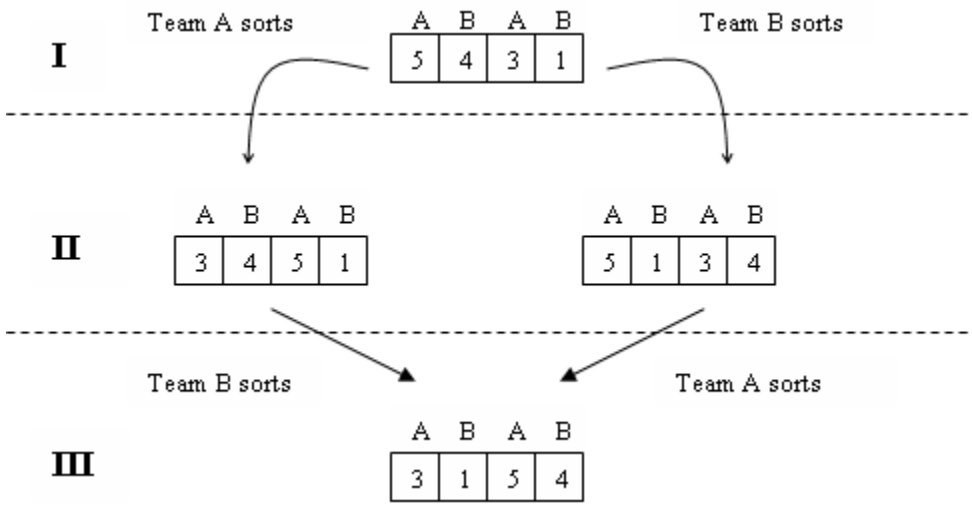

Figure 5.1. Ambiguous Inversion Removal of (4,3)

One approach is to distribute credit equally among all teams that could have possibly removed or introduced the inversion in question. In addition to being algorithmically difficult, this technique has several disadvantages in terms of theory and validity. Whether a swap of elements occurs or not is strongly influenced by a team's experience; conversely, the outcome a swap influences whether a member of the team leaves the company or not, which affects the team's experience. This linkage is severely weakened if task responsibility is diluted over several teams. Also, this approach does not have a clear analog in the real world. One could argue that such ambiguous cases represent tasks that cannot be decomposed to the team level, but instead are tasks requiring input from multiple teams. Due to the random nature of the simulation, however, an inversion removal that is ambiguous during one replication may not be ambiguous during another even if the original requirements array is seeded identically. This situation could arise, for example, if an inversion is removed during the first iteration of one replication but is passed over until the last iteration in another replication due to a mistake made by some team. A superior analogy relates ambiguous tasks to the representation of dependencies among components.

When a system is decomposed during requirements elicitation, dependencies among behaviors become apparent. In RUP, system behaviors are modeled as use cases (Krutchen, 1999), and dependencies as identifying relationships (Bruegge and Dutiot, 2000). Regardless of whether they share an identifying relationship, two use cases can be processed independent of one another until software components are integrated. By completing a seemingly isolated assignment, a team can fulfill requirements never

specified in the assignment itself. Put simply, software is more than the sum of its components.

A second approach to dealing with ambiguous inversion removal and introduction allows us to model this aspect of software development. In Figure 5.1, assume the left-hand path represents the event that actually occurs. Considering its assignment in isolation, team A fulfills the requirement {(5,3)}. At the project level, however, it contributes to the fulfillment of the requirement {(5,3), (5,4)}. This sorting action, however, may not place all elements of the set in their final locations. A use case may have dependencies, and the team has just implemented the use case (feature) assigned to it. At a later time, other teams or the same team will implement other features that will result in an array where all elements end up in their final locations. Until that time, the array is not completely sorted. Hence, the implementation is not complete. With respect to team A, the important point to note, however, is that team A exerted no effort (and may have had no knowledge of) completing requirement {(5,4)}. Therefore, this requirement should not factor into team A's performance rating. The Team-RUP framework ignores ambiguous inversions when team performance is evaluated.

By ignoring ambiguous inversions, we can know exactly how many (unambiguous) inversions a team should have removed during any particular iteration. Suppose the requirements array is to be *n*-sorted by *k* teams during iteration *i*, and *m* inversions should be removed by this operation. The expected number β_{ti} of removed inversions for team *t* is defined as follows:

$$\beta_{ti} = \frac{m}{k} \tag{5.1}$$

Assuming team t actually removed α_{ti} inversions during iteration i, the milestone performance δ_{ti} of t during i is define by Equation 5.2:

$$\delta_{ti} = \frac{\alpha_{ti}}{\beta_{ti}} \tag{5.2}$$

While milestone performance is measured each iteration, team performance is computed over a *window* consisting of multiple iterations, and involves a moving average of the milestone performance measures observed during these iterations. This moving average mitigates the effects of variance, thereby providing a more accurate indicator of typical performance. For example, a team which does little during earlier iterations but accomplishes a great deal just before team evaluations will not fair as well as a team which works adequately but steadily. A team of the latter type represents less risk to a development organization and should be valued more highly. In addition to the moving average, a record of past shortcomings factors into the performance criteria and further diminishes the merit of short bursts of team effort.

Since β_{ti} represents a statistical expectation, Equation 5.2 should on average equal 1 in the case that teams have an equal chance of removing any particular inversion. Since assignments are distributed to teams randomly, the only factor disrupting this "equal chance" criterion is the quality of the team. For a less competent team, δ_{ti} will on average be less than 1. To exploit this information, we define the failure history f_{ti} of a team t for iteration i as follows:

$$f_{ti} = \begin{cases} 0 & \delta_{t(i-1)} \geq 1; \\ f_{t(i-1)} + 1 & \delta_{t(i-1)} < 1. \end{cases} \tag{5.3}$$

For a performance window of size w, we define the team performance P_{tk} of team t at the end of iteration k via Equation 5.4:

$$P_{tk} = \frac{1}{w} \sum_{i=k-w+1}^{k} \frac{\delta_{ti}}{f_{ti} + 1} \tag{5.4}$$

Performance evaluations take place every $w/2$ iterations providing the aforementioned moving average of milestone performance measures. In the current implementation, performance evaluations occur every two iterations and are based on four milestone performance measures (i.e., w=4).

During a performance evaluation, teams with a performance measure greater than a predefined threshold (0.3 in the current implementation) are immune to employee turnover. Of the teams below this threshold, the team with the smallest performance rating undergoes subjective appraisal. As evinced by Equation 5.4, team performance is purely an objective matter. In the real world, however, perceived performance is of far greater consequence and is subject to human error.

When a real-world team is evaluated, the evaluator has incomplete information concerning a team and is influenced by various biases. To reflect this fact, the separation of an employee from the company in the Team-RUP framework is a probabilistic event. The probability F_{ti} that some member of team t will separate from the company at the end of iteration i is given by the following equation:

$$F_{ti} = B + S \cdot (1 - P_{ti})^3 \tag{5.5}$$

Notice that F_{ti} is computed for at most one team per performance window. Also, the cubed term in Equation 5.5 ranges between 0.343 and 1 since P_{ti} lies in the interval from 0 to 0.3. If a team is eligible for subjective appraisal, two uniform random variates u_1 and u_2 are sampled. If $u_1 < F_{ti}$, a single member of team t leaves the company, and an additional variate u_3 is sampled. If $u_3 < 0.25$, the team leader is the member who leaves; otherwise, it is one of the other members. If the leader leaves, the team undergoes a reorganization period in which productive work ceases. A new leader is assigned to the group, and the team as a whole adjusts to the new circumstances. In Team-RUP, this situation corresponds to a stall period of three days in which the team remains idle. In addition, the experience of the team decreases by the following amount:

$$1.2 \cdot (0.25 + 0.25u_2)$$

In contrast, the experience of the team decreases by a lesser amount if the departing member is not the leader:

$$0.25 + 0.25u_2$$

Moreover, the team does not stall in this case. If no member of the team is fired, the experience of the team is *increased* by an amount dependent on the team's current experience e_c:

$$u_2 \cdot \frac{1 - e_c}{2}$$

From the above discussion, we see that internal turbulence can be influenced by the researcher by adjusting the base rate and sensitivity to performance parameters. A

strict interpretation of Equation 5.5 requires these two parameters to sum to 1 in order to satisfy the definition of a probability. If we assume all values outside the interval [0,1] form an equivalence class with the nearest endpoint, however, this requirement can be bypassed. For example, a value of 2 for F_{tk} is interpreted as a certain event. Internal turbulence T can then be quantified as any linear combination of the base rate and sensitivity to performance:

$$T = \alpha B + \beta S$$

In the current implementation, α and β are set to 1 and B=S.

CHAPTER 6:

Programmed Model

The Team-RUP framework is implemented in Java 1.5. As described in Chapter 4, agents are implemented as collection of objects. No agent has a programmatic reference to any other agent. Instead, each agent is endowed with an address book containing mailbox numbers of all the agents in its acquaintance network. Agents do have references to the post office through which messages are sent and received. For each agent, checking the mail is the event that starts most activities. Agents can retrieve a single message from their mailboxes once per time tick. Therefore, the duration of activities triggered by such events must be a carefully considered component of Team-RUP.

1. Deadlines & Timing

In the Team-RUP model, we make the assumption that one workday equals 15 time ticks. Therefore, a month of five-day work weeks maps to approximately 300 ticks. RUP iterations typically range from one to three months. Because the scope of a project has the greatest impact on the time it takes to complete the project, the following restrictions are made. A project with a very small (low) scope is allowed 300 ticks (one month) per iteration. Iterations for projects with small or medium scope are allotted 600 ticks (two months). Finally, projects with large or very large scope are allowed 900 ticks (three

months) per iteration. To maintain configuration integrity, the deadlines associate with an iteration may "slip" by a small number of ticks. An absolute deadline equal to the number of iterations times the time-box length is set for the entire project. The simulation ends regardless when this deadline is reached. While the scheduling and execution of events is carried out by classes unique to the Team-RUP implementation, an open source simulation engine developed at the University of Chicago controls the event calendar.

2. Repast

Management of the schedule is handled by the Recursive Porous Agent Simulation Toolkit (Repast 3.0). The Team-RUP model extends a basic model class of Repast and makes use of its visualization components for verification. Also, events subclass Repast objects intended for this purpose. The agents, communication system, work medium, and utility components are assembled from Java classes external to the simulation toolkit.

3. Verification

Jerry Banks (1998, p. 336) defines verification as, "substantiating that the model is transformed from one form into another, as intended, with sufficient accuracy." Verification of the Team-RUP programmed model involved several steps. Each class was subjected to a code review immediately after it was programmed. Similarly, code reviews were performed on each package upon its completion. A final code review was performed on the entire model after it was completely implemented. To supplement the review process, thorough testing was performed on the compiled project.

Testing took place on two levels. Pictured in Figure 6.1, a graphical component was implemented to verify that a modified Shell sort was actually occurring.

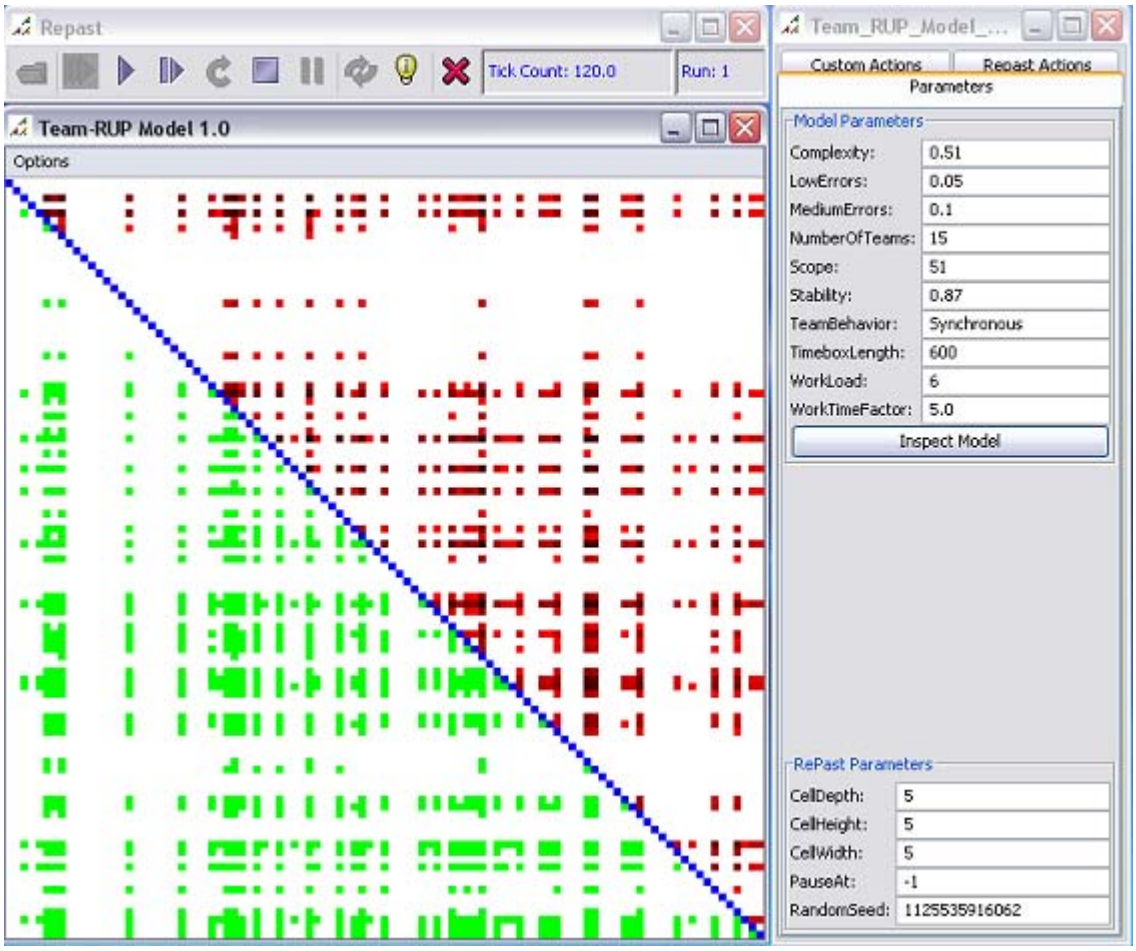

Figure 6.1. Graphical Depiction of List Inversions

In Figure 6.1, each ordered pair present in the requirements array is plotted in accord with a rectangular coordinate system in which the origin lies in the upper left-hand corner of the graph. The abscissas increase from left to right and the ordinates increase from top to bottom. Each green pixel represents two correctly ordered array elements. Elements out of order are represented by red pixels. The shade of the red pixels corresponds to the

distance between ordered pair elements in the requirements array. Two elements that are far apart in the array appear in a darker shade than those appearing closer together. From Chapter 3, Section 2.2, we see that darker pixels correspond with more risky requirements. The blue pixels are always present in the display and correspond to pairs with equal elements.

By viewing this display while simulations were running, we could confirm several model requirements. First, the requirements arrays are actually being sorted. Second, inversions far apart in the array are being removed in early RUP iterations. This feature reveals a primary characteristic of Shell sort. Another Shell sort characteristic revealed through the display is its iterative "semi-sorting" approach. As the simulation progresses, the red pixels migrate toward the blue line. Thus, inversions involving numbers that differ by a small amount are the last inversions to be removed. A final verification point confirmed by the display is the lack of duplicates in the requirements array. While not vital to Team-RUP's conceptual design, this feature provides a useful method for tracking which team is doing a particular activity over some duration of time. That array elements are distinct is verified by observing that the blue pixels never change to green pixels. Another important part of verification involves the coordination of teams.

In the Team-RUP framework, teams coordinate according to protocols reflecting the sorting algorithms discussed in Chapter 4, Section 1. Since these protocols are the principle modeling component for team behavior, it is paramount that they be implemented correctly. For each team behavior, driver programs were developed that printed a notification of each agent action as it occurred. This text allowed us to confirm that teams with the appropriate qualifications were managing the correct array locations

and elements at the appropriate times during simulations. Ensuring that these protocols actually encoded the desired team behaviors was part of validation.

4. Validation

To validate the Team-RUP implementation, we must show that the model "…behaves with satisfactory accuracy consistent with the study objectives" (Banks, 1998, p. 336). The objective of this study is to research the efficiency and effectiveness of software development teams with archetypal behaviors that define the degree of autonomy in collaboration and the degree of concurrency in coordination. Because of its widespread advocacy and following among software professional today, we are particularly interested in the Agile behavior and whether or not its validity is computationally verifiable. With these three objectives in mind, we developed three validation tests.

The first test focuses on the degree of concurrency in coordination. Whenever multiple tasks and multiple labor sources co-exist, the issue of synchrony arises. It is a topic common to the social sciences, business, and engineering. Typically, asynchronous systems are implemented with the hope of improving performance. Depending on the task at hand, however, this strategy may or may not be effective. Performance, therefore, can not be used to distinguish between linear and concurrent systems. Stability, however, can be used as a delineating factor.

Asynchronous systems are less predictable than synchronous systems (Shamsi, Chu, and Brockmeyer, 2005). This lack of stability results from the fact that asynchrony provides for a greater number of potential scenarios. For example, consider two software developers implementing four components. In one scenario, the developers complete

work at the same pace and address two components each. In another scenario, one developer finishes work more rapidly than the other and completes three of the four components. These two situations could give rise to drastically different process measures. If the developers addressed components in a linear fashion, however, the manner in which tasks are completed is not affected by the performance of the individual developer. This reduces variability among output statistics. It is important to note that this is not necessarily a value judgment concerning synchrony. Rather, it isolates an inherent trait that can be validated against.

To validate the Team-RUP programmed model in terms of coordination, we designed a test quantifying predictability in terms of standard deviation. This test utilizes a 5^2 factorial design defined by Table 6.1 (Law and Kelton, 2000).

Level	Workload	Number of Teams
Low	2	2
Moderately Low	3	3
Moderate	5	5
Moderately High	7	7
High	9	9

Table 6.1. Factor Coding, coordination validation experiment

With each behavior type, 50 replications were run for each factor combination. Productivity, staff utilization, timeliness, and quality measures were collected. For each of these metrics, we calculated the standard deviations across each set of replications. This calculation resulted in 25 data points per metric per team behavior type. The average of each set of 25 data points was calculated and histograms were generated (Figures 6.2-6.5).

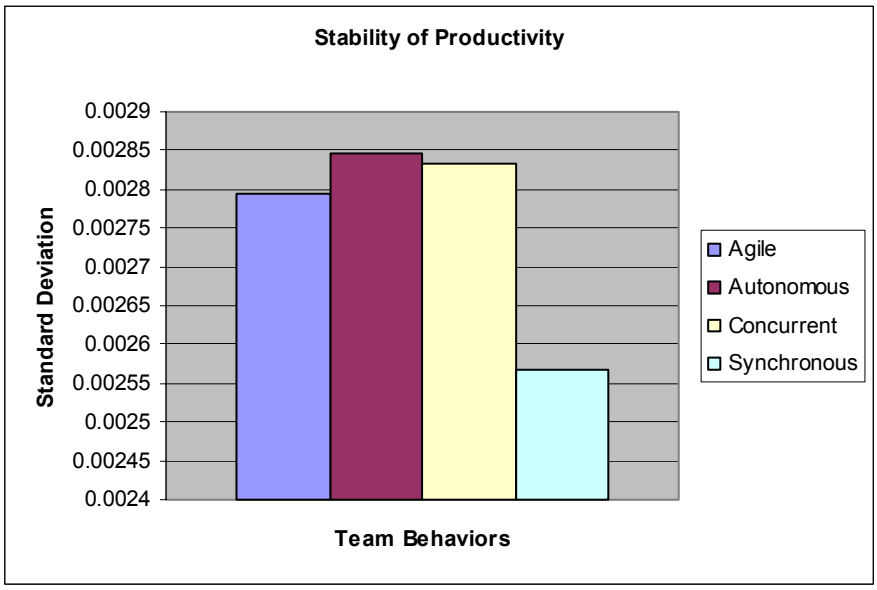

Figure 6.2. Predictability of Team Productivity

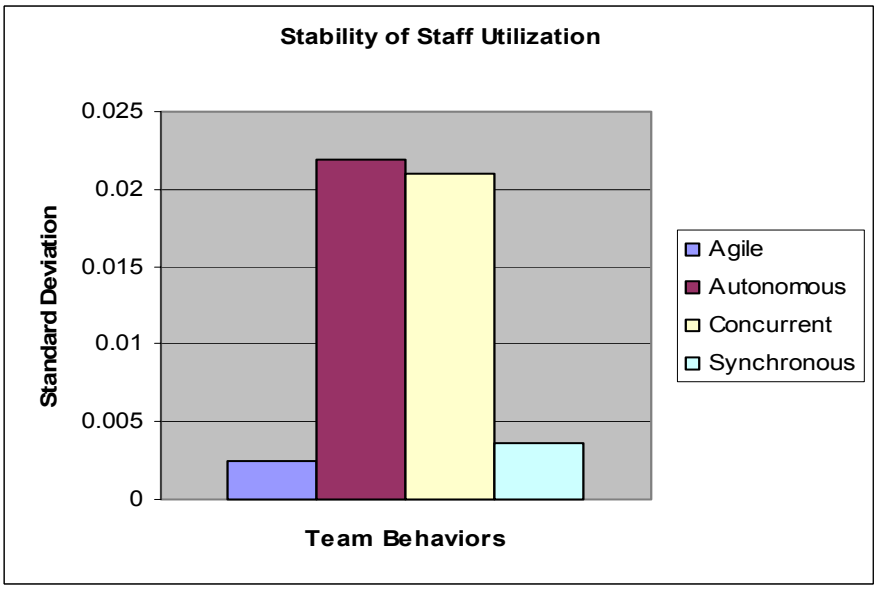

Figure 6.3. Predictability of Staff Utilization

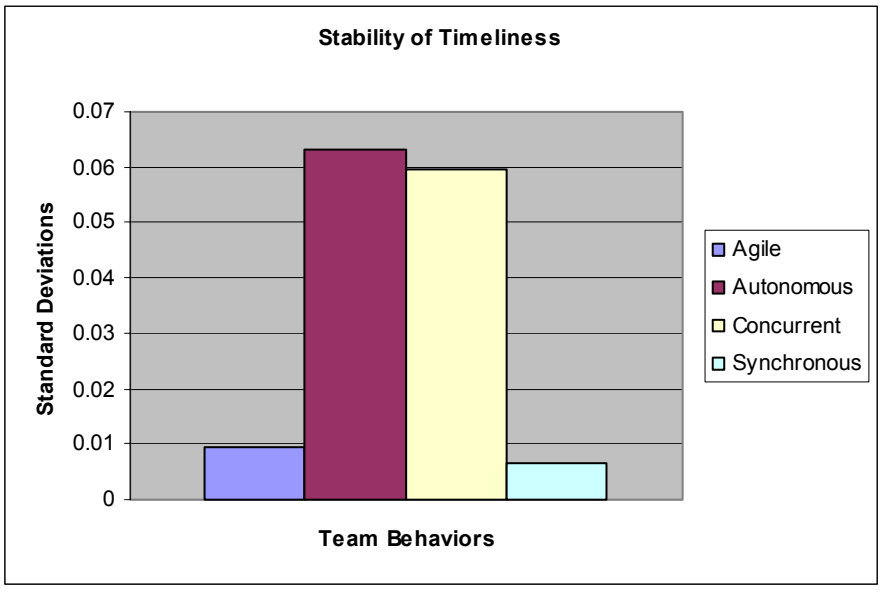

Figure 6.4. Predictability of Timeliness

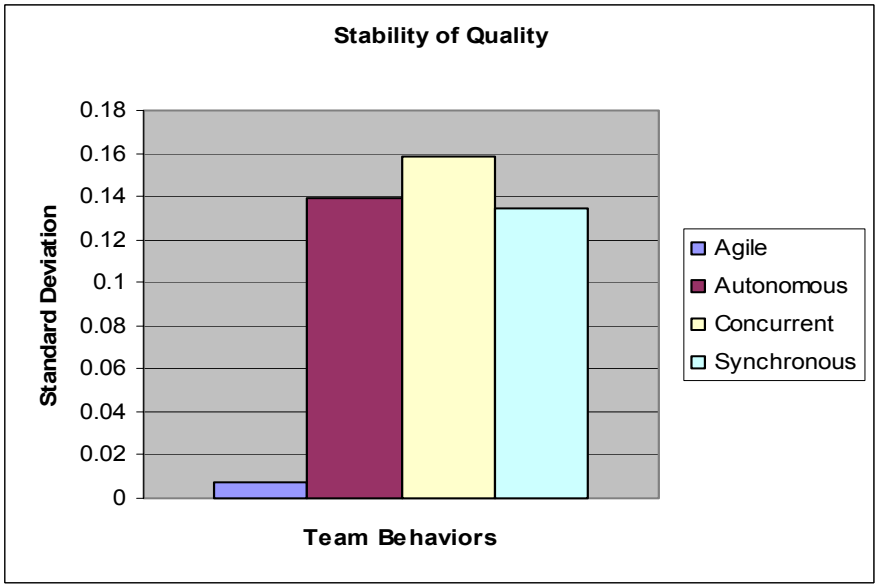

Figure 6.5. Predictability of Quality

In each of these graphs, Autonomous and Concurrent behaviors exhibit greater variability than Agile and Synchronous teams. That is, teams that coordinate their work efforts concurrently are less predictable than those that synchronize their activities.

Our second test focuses on the degree of autonomy in collaboration. That is, it focuses on the delineating characteristics of teams that use top-down versus bottom-up collaboration strategies. Top-down strategies offer a tighter fit between the set of requirements and the implemented product. They are more intuitive and match well with strategies employed by other engineering disciplines. Unfortunately, they respond poorly to requirements change. A new development plan must be created or the original plan must be adapted. Because implemented components are delayed until the end of the project, strict deadlines may result in unfulfilled requirements. Bottom-up strategies reverse this trend. Though less intuitive, a bottom-up strategy is more adept at responding to change (Pizka, M. and Bauer, A., 2004). Rather than affecting the entire project, the impact of requirements change is limited to those components that have already been implemented and which fail to meet the new requirements. Thus, bottom-up teams are still able to complete a large percentage of the requirements even in the face of changing requirements. For high levels of external turbulence, however, this difference diminishes.

To validate the Team-RUP programmed model in terms of collaboration, we designed a test involving five levels of stability. Note that lower levels of stability correspond to higher levels of requirements change, whereas higher levels of stability correspond to lower levels of requirements change. These levels are defined by Table 6.2. For each factor level, 50 replications were run. The average quality measure across each set of replications was calculated. Recall that quality records the fraction of

requirements completed. Figures 6.6 and 6.7 reveal the results of this test for each team behavior.

Level	Stability
Low	U(0.0,0.2)
Moderately Low	U(0.2,0.4)
Moderate	U(0.4,0.6)
Moderately High	U(0.6,0.8)
High	U(0.8,1.0)

Table 6.2. Factor Coding, collaboration validation experiment

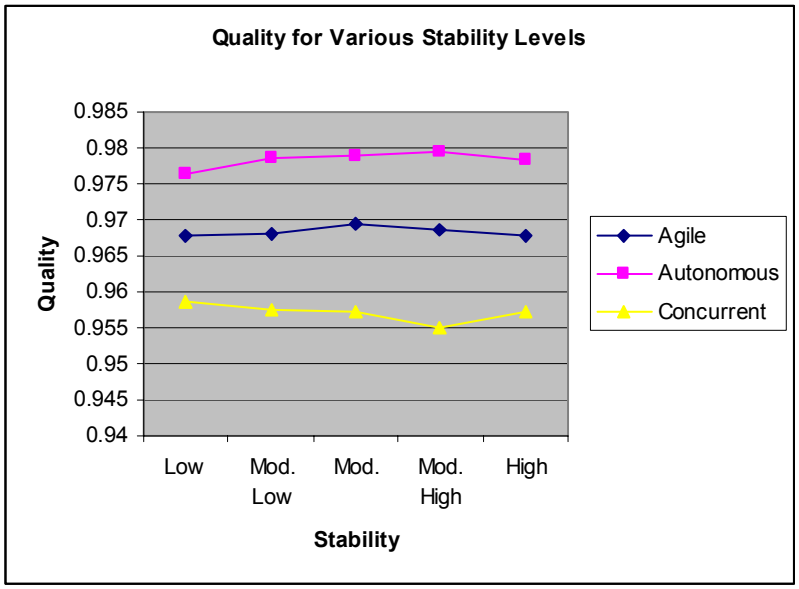

Figure 6.6. Quality for Agile, Autonomous, and Concurrent Teams

We place Synchronized teams in a separate graph because the quality level is so much lower than it is with the other behaviors. Each of the aforementioned validation points is supported by these diagrams. The Autonomous and Agile teams cope better with requirements change than Concurrent and Synchronized teams. Especially for the teams appearing in Figure 6.6, the differences are less pronounced at lower levels of stability.

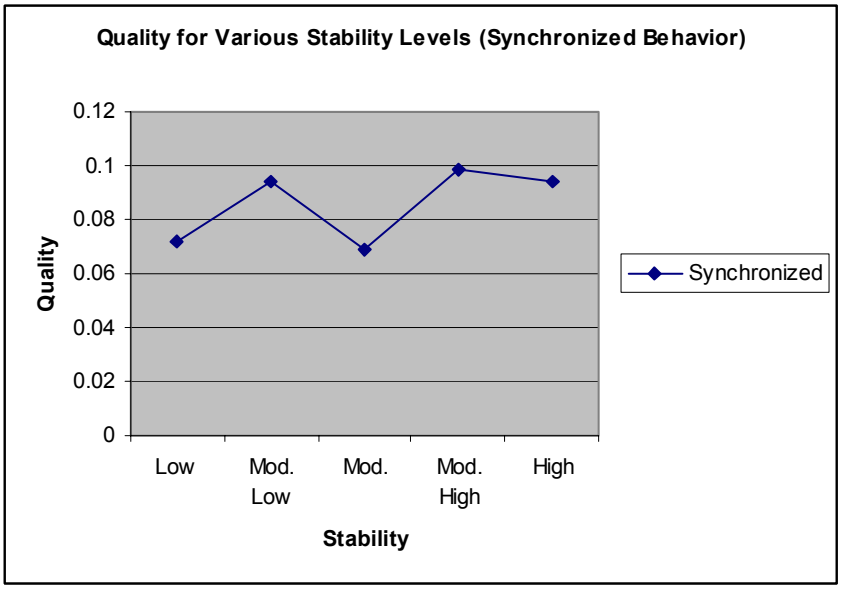

Figure 6.7. Quality for Synchronized Teams

Our final validation test centers on the Agile team behavior. Though the Team-RUP behavior taxonomy differs significantly from that proposed by Constantine (1993), a high degree of similitude exists between Agile and Random teams. Both emphasize innovation and posture themselves as the antithesis of the traditional hierarchy. We therefore rely on the following characteristic of Random teams to test the validity of Agile teams: Parent organizations absorb the inefficiencies of Random (Agile) teams and shield them from fluctuating market trends; therefore, efficiency is not an inherent trait of Random (Agile) teams (Constantine, 1993). An efficient team accomplishes more work in less time with fewer people. Therefore, efficiency can be quantified as the ratio of productivity to staff utilization.

To validate the Team-RUP programmed model with respect to Agile teams, we utilized a 5^2 factorial design defined earlier by Table 6.1. For each factor level combination, 50 replications were run. Efficiency measures were calculated for each

replication. Averages were computed across each replication set. Each of these averages is represented by a single point in Figure 6.8.

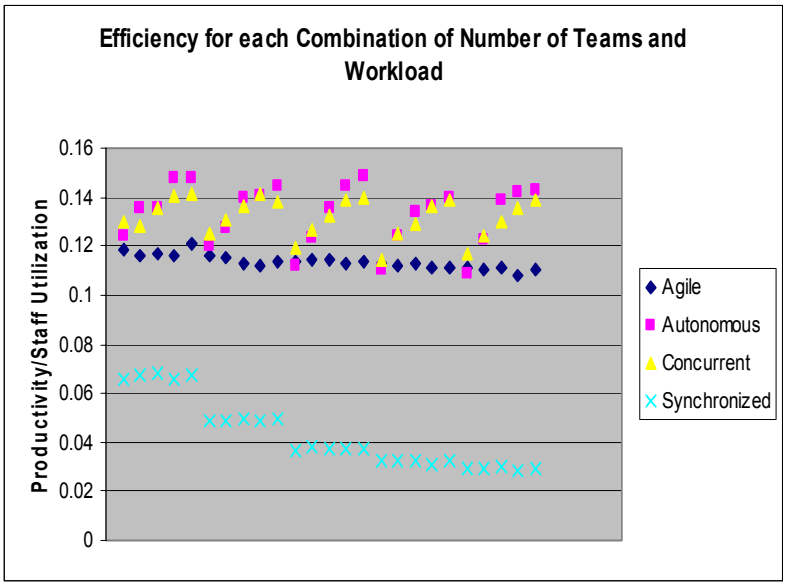

Figure 6.8. Inefficiency of Agile Teams

Note that labels on the horizontal axis have been omitted due to space limitations. These labels correspond to combinations of level factors and do not carry any particular significance. The important point illustrated by this figure is the fact that Agile teams are not particularly efficient in comparison to other team behaviors. We will see in the experimentation section, however, that Agile teams more than compensate for this deficiency in their ability to operate amidst turbulence: a fact of life in the real world.

CHAPTER 7:

Experimental Model

Because Team-RUP is an extensible framework rather than a narrowly-defined simulation model, the opportunities for experimentation are vast. In this initial study, however, we considered two questions that are of particular interest to both academia and industry today.

1. Agility Test

To address a very broad audience, we considered the issue of whether or not the claims of the agile process community are valid. Are better results achieved from adhering to principles such as those of the Agile Manifesto? For example, should software developers strive for "…early and continuous delivery of software" (Agile Alliance, 2001): a goal that requires at least a degree of bottom-up development? Can developers work in virtual independence (i.e., concurrent coordination) or is the following true: "Business people and developers must work together daily throughout the project" (Agile Alliance, 2001)? Finally, is it wise to "welcome changing requirements, even late in development" (Agile Alliance, 2001)?

1.1. Experiment Design

In this experiment, we compare the four team behavior in terms of their ability to cope with the combination of internal and external turbulence. Levels for these two factors appear in Table 7.1.

Level	Internal Turbulence	External Turbulence
Low	U(0.5,0.7)	U(0.0,0.2)
Moderately Low	U(0.7,0.9)	U(0.2,0.4)
Moderate	U(0.9,1.1)	U(0.4,0.6)
Moderately High	U(1.1,1.3)	U(0.6,0.8)
High	U(1.3,1.5)	U(0.8,1.0)

Table 7.1. Level Definitions for Internal and External Turbulence

Rather than test every combination of internal and external turbulence, we define levels for aggregate turbulence. The level definitions for this "fuzzy" factor are defined by Table 7.2 in terms of internal and external turbulence.

Internal / External	Low	Moderately Low	Moderate	Moderately High	High
Low	low	moderately low	moderate	moderately high	high
Moderately Low	moderately low	moderately low	moderate	moderately high	high
Moderate	moderate	moderate	moderate	moderately high	high
Moderately High	moderately high	moderately high	moderately high	moderately high	high
High	high	high	high	high	high

Table 7.2. Level Definitions for Turbulence

For each turbulence level, we run 50 replications. In each replication, a white cell from Table 7.2 with a corresponding turbulence level is randomly selected. The row and

column headings for this cell are used as the factor levels for internal and external turbulence during the replication. Metric averages for productivity, timeliness, staff utilization, and quality are taken across replication sets.

1.2 Results

Figure 7.1 records the productivity of teams with each behavior type under the various levels of turbulence.

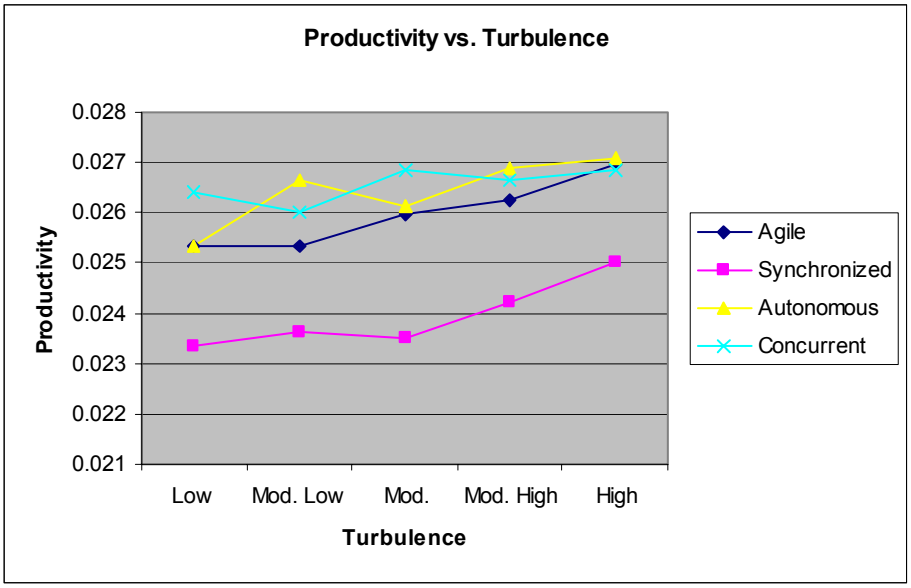

Figure 7.1. Productivity Under Various Turbulence Levels

The staff utilization of these teams is pictured in Figure 7.2.

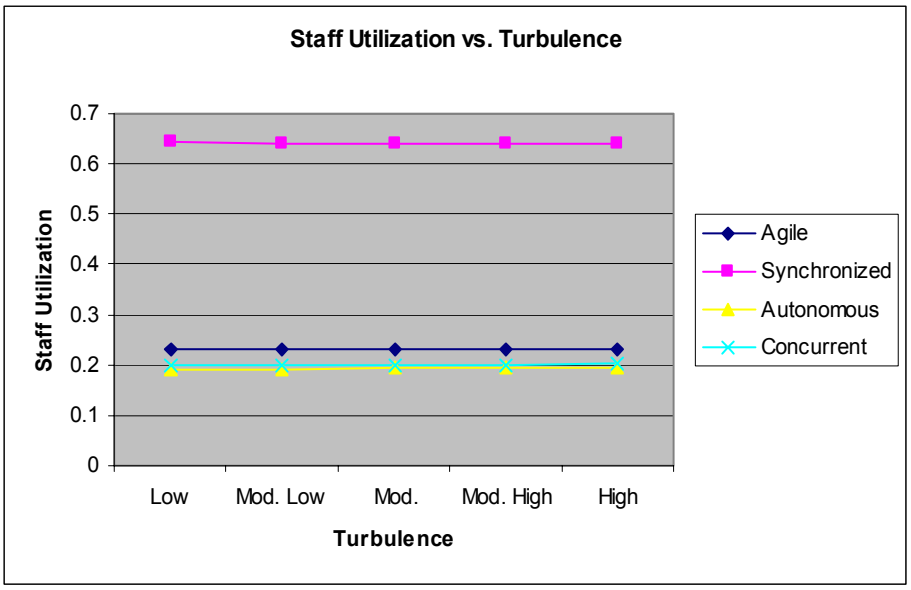

Figure 7.2. Staff Utilization Under Various Turbulence Levels

Figure 7.3 reveals the timeliness results of team behavior type.

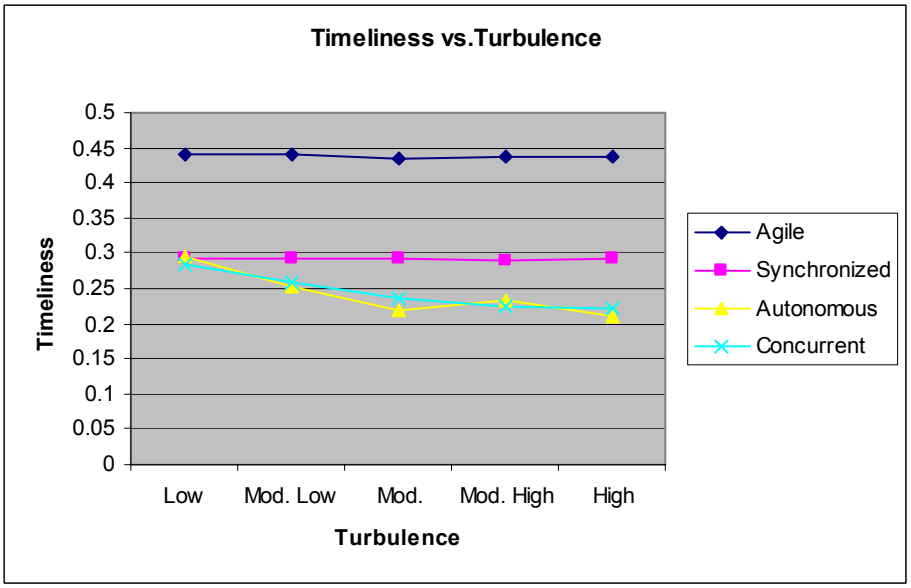

Figure 7.3. Timeliness Under Various Turbulence Levels

Finally, the quality results for teams of each behavior type are shown in Figure 7.4.

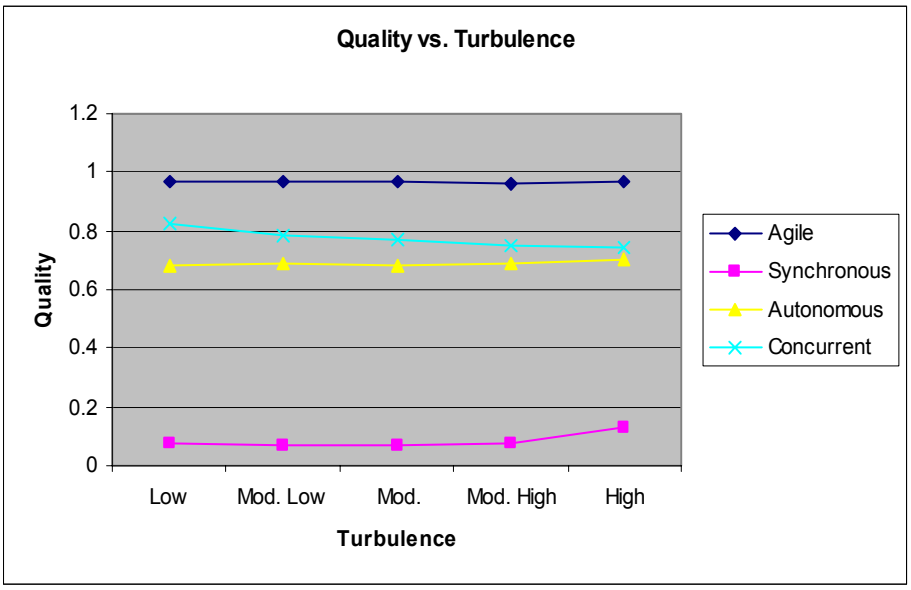

Figure 7.4. Quality Under Various Turbulence Levels

We now consider the implications of these graphs.

1.3 Discussion

Figures 7.2-7.4 provide some very useful insights into the nature of teams operation under turbulence. Note for example the linear, flat nature of each of the series. This trend reveals the insulating nature of the Rational Unified Process. Though each team behavior is drastically different, all are able to establish bounds on the negative effects of requirements change and employee turnover. Productivity adheres to this trend the least among the metrics; however, an increase in productivity can hardly be seen as a negative result. Moreover, this direct relationship between turbulence and productivity makes perfect sense. To meet deadlines in the presence of change, teams must accomplish more in a shorter amount of time. It is also interesting to note which team type adheres to this pattern the least.

In every graph but Figure 7.2, the series for the Concurrent team behavior has a definite positive or negative slope. As mentioned in the previous paragraph, the

productivity of Concurrent teams increases as turbulence increases. At the same time, the timeliness and quality of these teams decreases. This trend, however, should not be surprising. Concurrent teams work in a non-linear, top-down fashion. As discussed in Chapter 6, Section 4, this strategy is quite brittle with respect to change. Its top-down nature in particular leads to delays. Another interesting aspect of the graphs is observed when one compares the series to each other in absolute terms.

Each graph depicts either a clear "loser" or clear "winner". In every category but timeliness, the Synchronized behavior has a clear disadvantage in comparison to the other archetypes. This trend results from the fact that the Synchronized behavior is the most formal and rigid. Synchronized teams work in a top-down, synchronized fashion, which does not respond well to change. From Figures 7.1 and 7.2, we see that these teams are plagued by overhead. Even though the developers are exerting a great deal of effort, Figure 7.4 reveals that the quality of the final product still suffers. The most interesting observation that can be made from these graphs addresses the purpose of this experiment.

These results provide quantitative evidence that an agile approach to software development is the most effective strategy when coping with an unstable development environment. The Agile teams produce higher quality software and are more capable of meeting deadlines. From a customer's point of view, these are the only two factors that matter. A development firm, however, is also quite interested in productivity and staff utilization. Although they fail to surpass all the teams with other behaviors, it is evident from Figures 7.1 and 7.2 that Agile teams certainly hold their own. We conclude, therefore, that "agility" is a valid and useful counterbalance to the inevitable change involved in most software projects.

2. Team Size Test

Our second experiment addresses an important but often neglected segment of the development community: small businesses. In particular, we are interested in discovering how well small development organizations cope with turbulence. We suspect that larger organizations are better able to absorb the ill effect of turbulence. When requirements change or people leave the organization, the impact on smaller teams is felt on a global level. We wish to discover how different team behaviors fair with respect to varying levels of turbulence.

2.1. Experiment Design

In this experiment, we vary the number of teams under various turbulence levels. The turbulence levels and implementation are the same as in the first experiment. The levels of the number of teams factor are defined by Table 7.3.

Level	Number of Teams
Low	2
Moderately Low	3
Moderate	5
Moderately High	7
High	9

Table 7.3. Level Definitions for Number of Teams

For each level and team behavior, we run 50 replications. Metric averages for productivity, timeliness, staff utilization, and quality are taken across replication sets.

2.2 Results

We group the results according to metric. Figure 7.5 provides a legend for each graph.

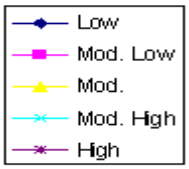

Figure 7.5. Series Legend for Results in Section 2.2

Figure 7.6 shows the productivity of each team behavior for various levels of the number of teams factor.

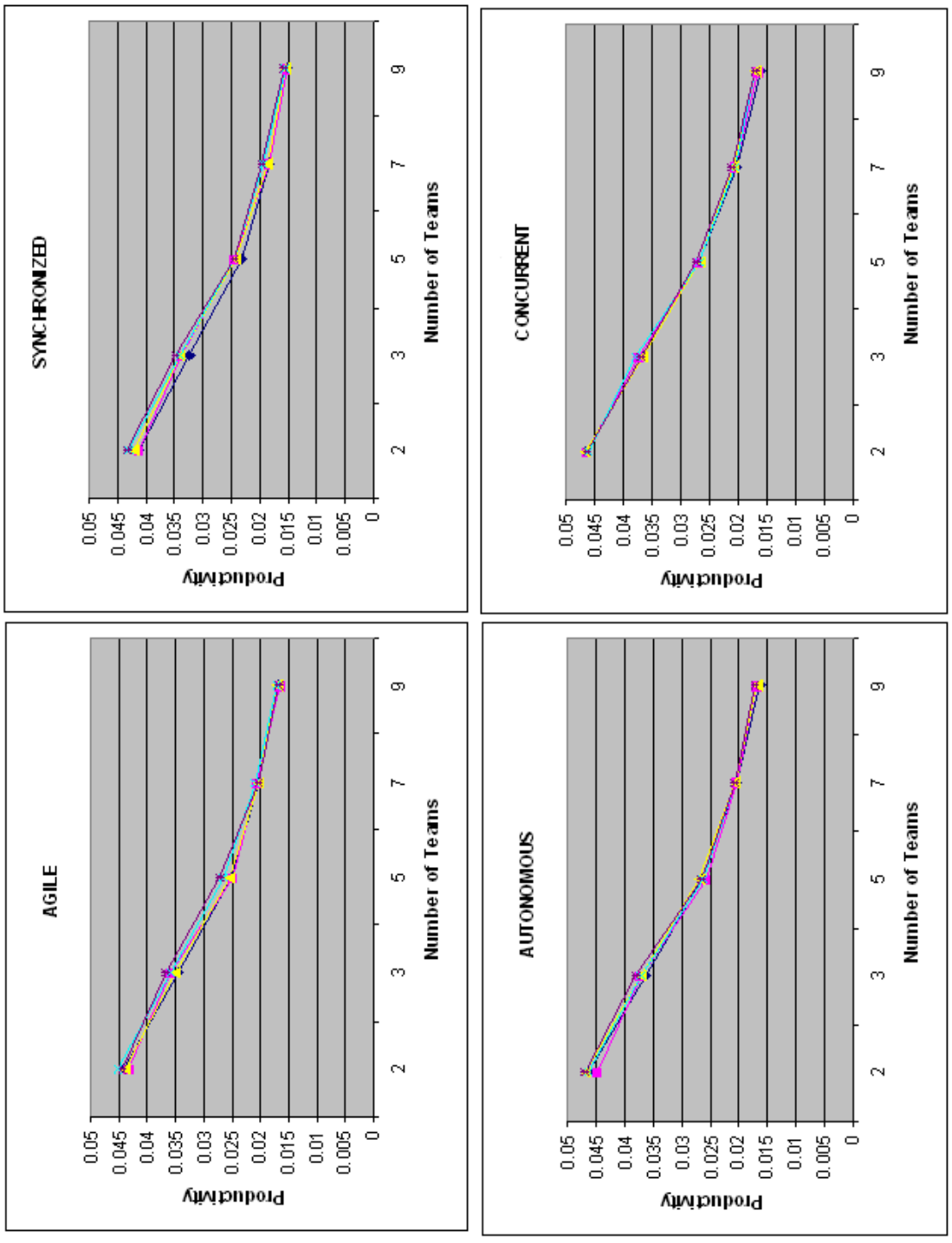

Figure 7.6. Productivity for Team Sets of Various Sizes

The staff utilization for each level of the number of teams factor is shown in Figure 7.7.

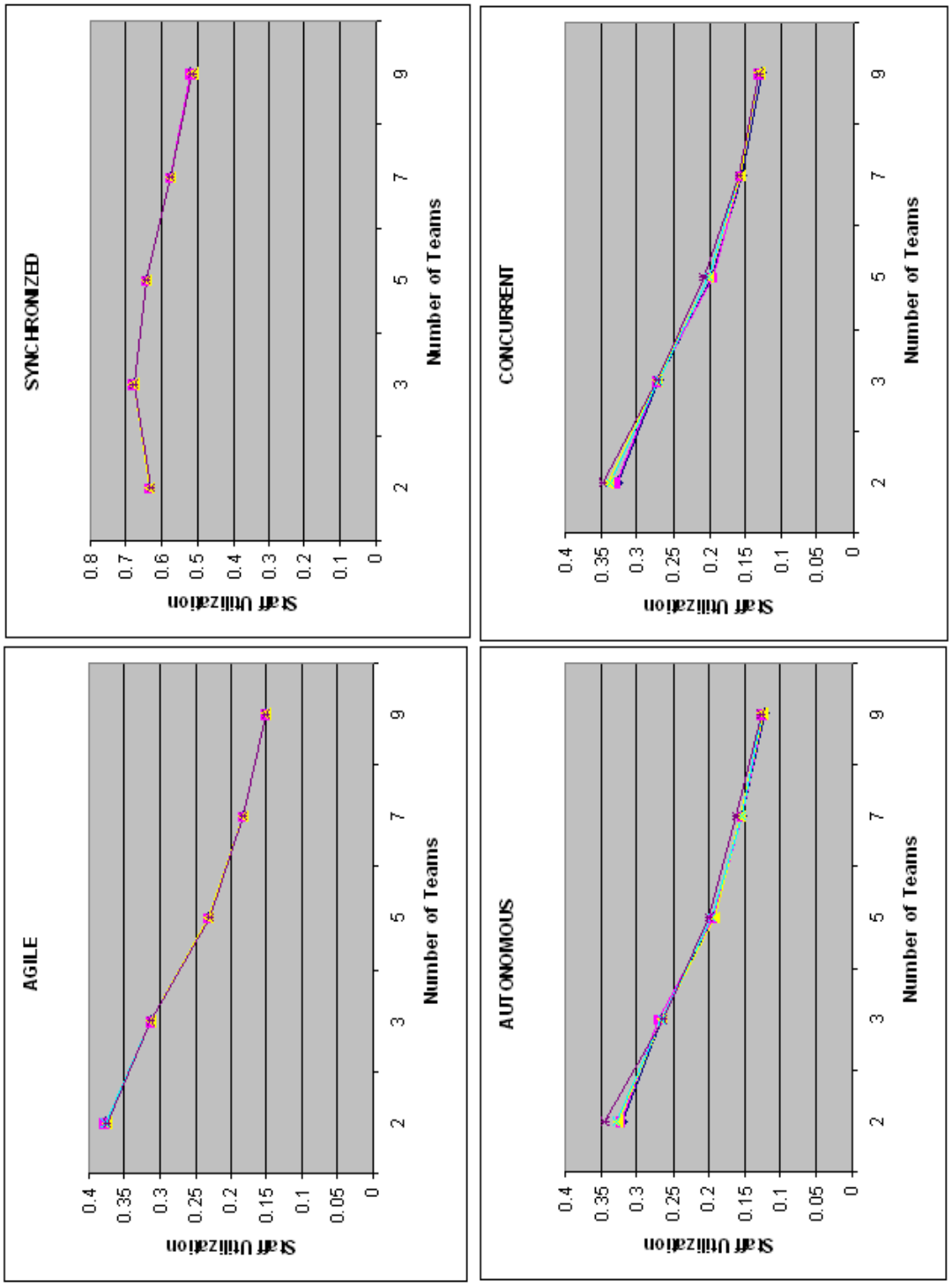

Figure 7.7. Staff Utilization for Team Sets of Various Sizes

Figure 7.8 reveals the results with respect to the timeliness metric.

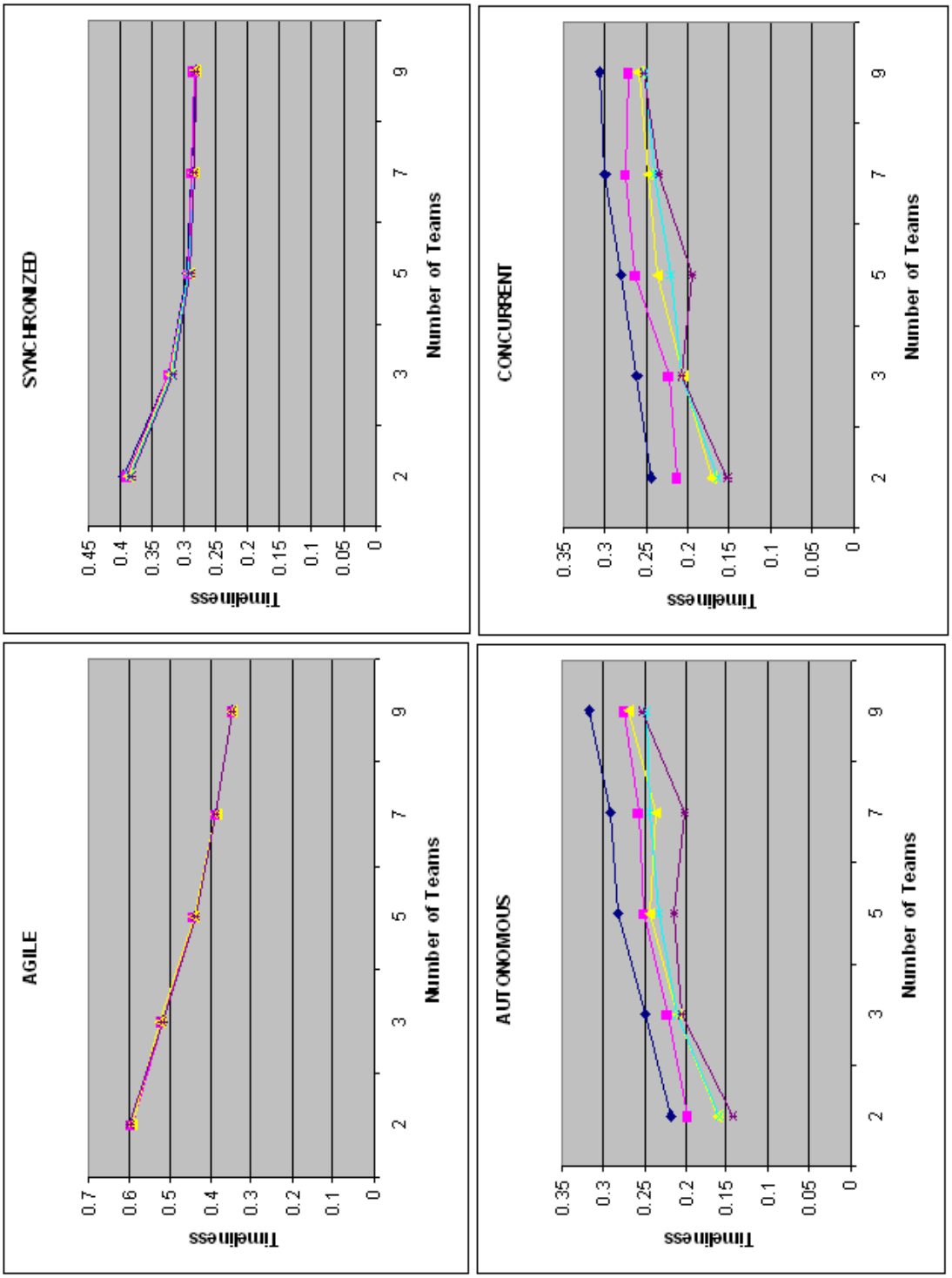

Figure 7.8. Timeliness for Team Sets of Various Sizes

Finally, the results for the quality metric appear in Figure 7.9.

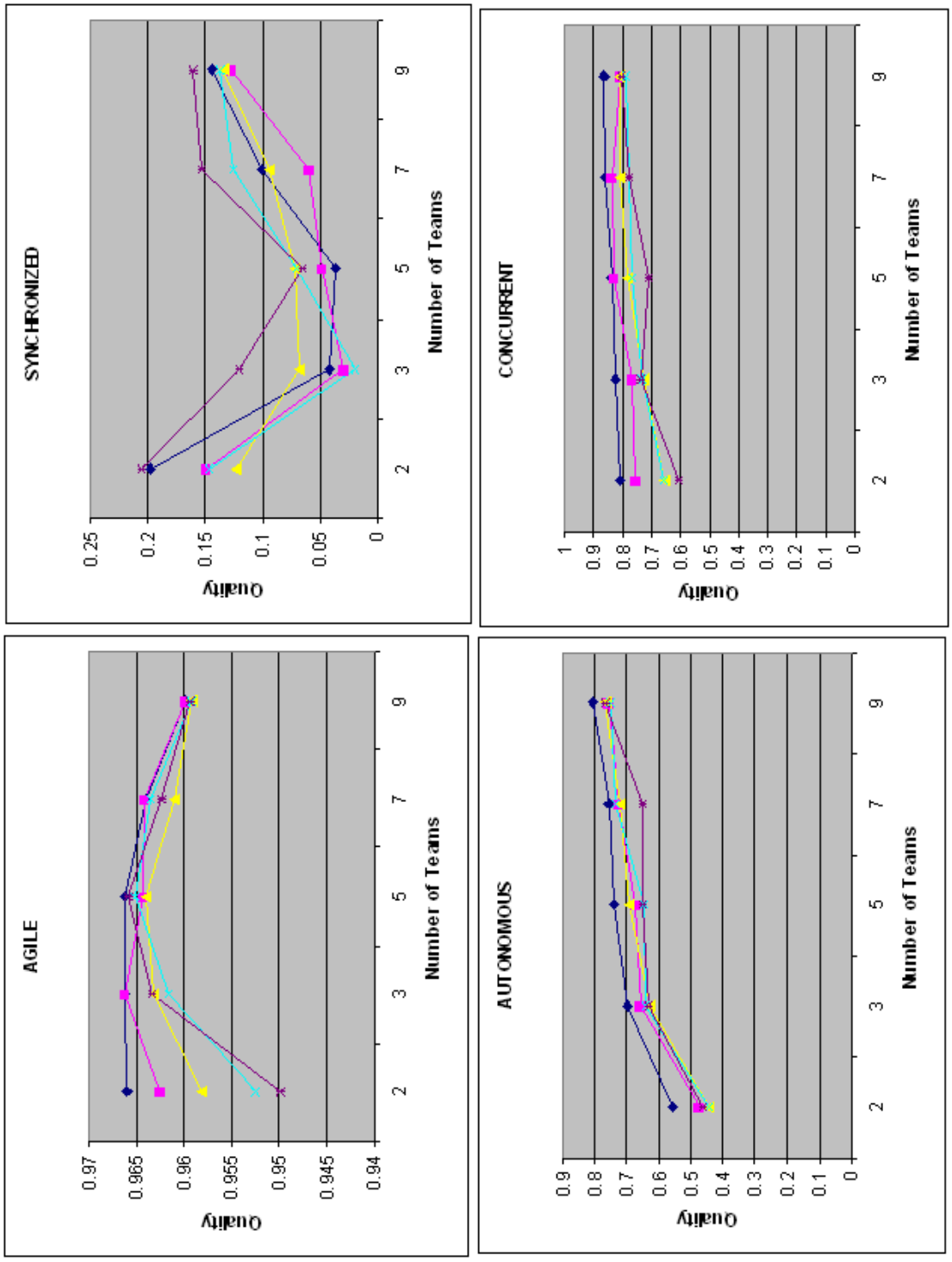

Figure 7.9. Quality for Team Sets of Various Sizes

2.3 Discussion

The results for this experiment reveal several interesting trends. In terms of productivity, all the teams perform similarly. As would be expected, increasing the number of teams leads to a decrease in productivity. More resources (teams) are expended on the same number of requirements. It is interesting to note that a point of inflection exists for each team and each turbulence level at some point between 3 and 5 teams. A similar trend exists for the staff utilization graphs.

Except for an initial increase for Synchronized teams, staff utilization also decreases as the number of teams increases. The initial utilization increase for Synchronized teams can be explained in terms of specialization. As the number of teams increases, the more specialized teams become. This increases the likelihood that a team will encounter a task with which it requires another team's specialized skills. Since more teams must be involved, staff utilization increases. This trend is confounded by the fact that adding teams creates more opportunities for idleness. Another interesting feature of Figure 7.7 centers on the variability across turbulence levels. The series lines for Agile and Synchronized teams overlap almost entirely, whereas those for Autonomous and Concurrent teams diverge. In the case of the latter behaviors, higher turbulence corresponds to slightly higher staff utilization. This trend is a result of the concurrent coordination strategy of these two team types. A team that finishes its assignment independently of the other teams may address newly added requirements, thereby raising the utilization. This split along coordination strategies is also observed in terms of timeliness.

Figure 7.8 reveals the first major differences between team behaviors in this experiment. As with staff utilization, the series lines are tightly grouped for Agile and Synchronized teams, whereas significant divergence exists between the series lines for Autonomous and Concurrent teams. This fact reveals that teams working in a linear fashion are less disrupted by change. Of greater interest to this study, however, is the fact that Agile and Synchronized teams are timelier when fewer teams are present. The synchronization tasks are less costly when fewer people are involved. In contrast, Autonomous and Concurrent teams are timelier when more teams are present. Because of the greater number of teams, more components can be distributed at any given time for concurrent implementation. The payoff associated with this staff increase is significantly reduced as turbulence increases. A final point of interest concerning Figure 7.8 center on the Agile team behavior. The Agile team behavior surpasses all other behaviors for every factor combination. At its worst setting of 9 teams, the Agile team is timelier than Autonomous and Concurrent teams. Figure 7.9 reveals that Agile teams face a tradeoff between timeliness and quality.

The quality graphs in Figure 7.9 display several interesting characteristics. Although the relationship between the number of teams and timeliness is linear for the Agile behavior, an optimal setting can be found for the former value in terms of quality. In particular, 5 teams yield the best quality response for Agile teams. For the Agile behavior, quality is less uniform over the turbulence levels than with other metrics. The trend is particularly evident when only two teams are present. This point supports our claim that smaller organizations are more deeply affected by change regardless of their process strategy. On the other hand, the difference between the best and worst quality

resulting from Agile teams is less than 2 percentage points. Furthermore, this worst quality rating surpasses the best generated by any other team. This fact reveals the superiority of the Agile behavior with small organizations. Another interesting tradeoff can be found with Synchronized teams.

Figure 7.9 further condemns the Synchronized strategy. This most rigid strategy produces low quality software for each setting. It performs its worst for teams of size 3 and 5. Recall from Figure 7.7 that these two levels corresponded to higher levels of staff utilization due to specialization. Specialization also explains the low quality responses at these levels. When many specialists are involved on a project, quality increases because worker skill sets are more finely honed. This observation is a simple restatement of Adam Smith's (1993 version) division of labor principle. We see this trend in Figure 7.9 with the use of 7 and 9 teams. For a small number of teams, however, the specialization areas are too large for skill improvement to lead to performance increase. However, the labor divisions prevent teams from aiding one another, which creates blockages in the workflow. A final important aspect of Figure 7.9 centers on the teams that work in a non-linear manner.

The quality graphs for Autonomous and Concurrent teams closely match the corresponding timeliness graphs. That is, the utilization of more teams results in higher quality. Due to concurrent work efforts, a greater number of components can be addressed when more teams are present. Initial versions of components are produced more quickly and the time available for revisiting components is increased. Thus, deadlines are met more regularly and a greater number of requirements are fulfilled.

Based on Figures 7.8 and 7.9, we can therefore conclude that the Autonomous and Concurrent team behaviors are better suited for large, rather than small, enterprises.

CHAPTER 8:

Conclusion

Simulation has been an invaluable tool to researchers in a variety of disciplines. Its uses include what-if analysis, validation of empirical results, teaching complex skill sets, and exploring abstract relationships. Quite recently, software developers have employed simulation to the study of software processes in an attempt to stabilize an industry plagued by faultiness and an inability to meet deadlines. The multi-disciplinary nature of software development, however, has limited the scope of these efforts. The engineering of software lies at the confluence of computer science, sociology, and management science. Though uncommon in process simulation today, an agent-based approach allows for a more complete representation of the subject area.

Team-RUP is an agent-based simulation framework that supports the mutual study of software development, team dynamics, processes, organizational paradigms, and cultures. Used to create exploratory models, the framework is designed to discover general results applicable on an industry-wide basis. In order for these results to be useful in a real-world context, however, Team-RUP is firmly grounded in one of the most widely used process frameworks: the Rational Unified Process. Team-RUP uses a distinctive layered approach allowing features to be modeled at the individual, team, and enterprise level. Cross-cutting features like internal turbulence can also be captured.

Also, it uses a unique modeling approach based on sorting. Finally, inter-agent communication serves as a central modeling component.

In the current Team-RUP implementation, agency is represented on the team level. That is, the simulated development of software takes place on the level of interacting teams. A central feature of the Team-RUP framework is the classification of team behaviors according to the degree of autonomy in collaboration and the degree of concurrency in coordination. This classification yields four archetypical team behaviors: Agile, Autonomous, Concurrent, and Synchronous. In the current implementation, the efficiency and effectiveness of teams of these types under turbulent conditions have been of primary interest.

1. Result Summary

With the current implementation of Team-RUP, we have focused on two experiments. The first experiment centered on how teams with various team behaviors responded to different levels of internal and external turbulence. The second considered teams of various sizes in order to study the impact of turbulence on small development organizations. Each provided a plethora of interesting insights. First, we observed that the Rational Unified Process provides a layer of insulation against the negative effects of employee turnover and requirements change regardless of the team behavior type. However, Concurrent teams are most affected by turbulence. As turbulence increases, timeliness and quality decrease despite an increase in productivity. Although turbulence has the greatest impact on Concurrent teams, this behavior does not yield the lowest performance among the team types.

With its rigid structure, the Synchronized behavior is the least suited for adaptation to changing requirements and immovable deadlines. Division of labor among specialized teams can be either a help or a hindrance to Synchronized teams. When a large number of teams is present, the skill improvement due to specialization leads to increases in timeliness and quality. For a small number of teams, specialization can lead to blockages in the workflow, which ultimately reduces timeliness and quality. Standing in direct contrast to the Synchronized behavior is the Agile behavior.

This study provides quantitative evidence that agility is a valid and useful counterbalance to the inevitable change involved in most real-world software projects. If quality and timeliness are the primary objectives of a development effort, small organizations should strongly consider adopting an agile process to promote Agile team behavior. If, however, an Autonomous or a Concurrent approach is desirable for some project-specific purpose, it will be most applicable in larger organizations.

2. Future Work

The Team-RUP framework is highly extensible, adaptable, and modular. For example, each team agent could be replaced by a collection of agents, each representing some individual role. Each team could consist of designers, programmers, and testers. This addition would allow for the study of intra-team communication, collaboration, and coordination. To add these features would only require changing a single line of code in the current implementation. Furthermore, reducing the granularity of agents to the level of individuals would allow for the capture of individual emotions.

Human decisions and behavior are rarely derived solely from logical deduction. Also, inexperience is not the only reason developers make mistakes. Instead, an emotional dynamic influenced by a host of factors such as stress, trust, and competition often has a far greater impact on a development project. Team-RUP could be extended to encode these traits in individuals. This extension would be of particular value in studies of situations in which certain emotional states are expected. For example, stress levels could be elevated near a project deadline. Another potential extension centers on attaching meaning to the numbers within the requirements list.

In the current implementation, team performance is not directly affected by the value of the numbers being considered. It might be beneficial at some point to relate these numbers to skill sets or some aspect of risk management. For example, a team might be particularly adept at sorting inversions involving only even numbers but may not have all the skills needed to correctly sort inversions involving only odd numbers. Note that skill knowledge is quite distinct from general experience, a property already captured in Team-RUP.

Another extension to the Team-RUP framework would involve expanding the agent hierarchy to include the entire development organization. At present, the project manager, design manager, and construction teams are represented by agents. Requirements elicitation and testing are currently handled procedurally. To study employee interactions within these divisions, collections of agents could be developed to provide the functionality of the procedures.

Perhaps the most interesting addition to Team-RUP would involve the use of multiple requirements arrays. At present, we only consider a single development project

at a time. In the real world, however, development companies have multiple projects running concurrently. These projects compete for time and staffing resources. This competition could have a significant effect on the efficiency and effectiveness of an organization.

The Team-RUP framework represents a significant contribution to software process simulation in particular and software engineering in general. It promises far-reaching results grounded in a real-world process. Also, it adds a new technique to the toolbox of the simulation engineer: the list sorting model. Finally, it provides a plethora of outlets for extension. Formulated around the concepts of agency, requirements change, collaboration, and cooperation; Team-RUP provides a uniquely cohesive mapping between simulation reality and the manner in which software is truly engineered in the world today.

REFERENCES

Abdel-Hamid, T. and Madnick, S. (1991). *Software Project Dynamics: An Integrated Approach*. Upper Saddle River, NJ: Prentice Hall.

Agile Alliance. (2001, February 11-13). Manifesto for Agile Software Development (Principle Behind the Agile Manifesto). Retrieved February 26, 2006, from http://agilemanifesto.org/principles.html.

Aked, M. (2003). Risk reduction with the RUP phase plan. *The Rational Edge*. Retrieved January 30, 2006, from http://www-128.ibm.com/developerworks/rational/library/1826.html.

Albrecht, A. (1979). Measuring application development productivity. Proceedings of IBM Application Development Joint SHARE/GUIDE Symposium, Monterey, CA.

Axlerod, R. (1997). *The Complexity of Cooperation*. Princeton, NJ: Princeton University Press.

Banks, J. (1998). *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*. New York, NY: John Wiley & Sons, Inc.

Banks, J., Carson, J.S. II, Nelson, B.L., and Nicol, D.M. (2005). *Discrete-Event System Simulation* (4th ed.). Upper Saddle River, NJ: Prentice Hall.

Boehm, Barry. (1981). *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall.

Bruegge, B. and Dutoit, A. H. (2000). *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*. Upper Saddle River, NJ: Prentice Hall.

Charette, R. N. (1989). *Software Engineering Risk Analysis and Management*. New York, NY: McGraw-Hill.

Constantine, L. (1993). Work Organization: Paradigms for Project Management and Organization. *Communications of the ACM*, *36*(10), 35-43.

Donzelli, P. and Iazeolla, G. (2001). A Hybrid Software Process Simulation Model. *Software Process Improvement and Practice*. *6*, 97-109.

Ferber, J. (1999). *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. New York, NY: Addison Wesley Longman Inc.

Forrester, J. (1961). *Industrial Dynamics*. Waltham, MA: Pegasus Communications.

Fowler, M. (2004). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Boston, MA: Pearson Education, Inc.

Haberlein, T. (2004). Common Structures in System Dynamics Models of Software Acquisition Projects. *Software Process Improvement and Practice*. *9*, 67-80.

Kang, M., Waisel, L., and Wallace, W. (1998). Team Soar: A Model for Team Decision Making. In M.J. Prietula, K.M. Carley, and L. Gasser (Eds.), *Simulating Organizations* (pp. 23-45). Menlo Park, CA: The MIT Press.

Krutchen, P. (1999). *The Rational Unified Process: An Introduction*. Reading, MA: Addison Wesley Longman.

Krutchen, P. (2003). What is the Rational Unified Process? *The Rational Edge*. Retrieved May 5, 2005 from http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/feb03/WhatisRUP_TheRationalEdge_Feb2003.pdf.

Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Upper Saddle River, NJ: Prentice Hall.

Law, A. and Kelton, D. (2000). *Simulation Modeling and Analysis* (3rd ed.). New York: McGraw-Hill.

Leffingwell, D., and Widrig, D. (2000). *Managing Software Requirements: A Unified Approach*. Reading, MA: Addison Wesley Longman.

Lehman, M., and Ramil, J. (1999). The impact of feedback in the global software process. *Journal of Systems and Software*. *46*(2/3), 123-134.

Lucas, A., and Goss, S. (1999). The Potential for Intelligent Software Agents in Defense Simulation. *Proceedings of the 1999 Information, Decision, and Control symposium*, Adelaide, Austria, 579-583.

Martin, R. and Raffo, D. (2000). A Model of the Software Development Process Using Both Continuous and Discrete Models. *Software Process Improvement and Practice*. *5*, 147-157.

Mi, P. and Scacchi, W. (1990). A knowledge-based environment for modeling and simulating software engineering processes. *IEEE Trans. Knowledge and Data Engineering*. *2*(3), 283-294.

Myers, R. (1971). *Response Surface Methodology*. Boston, MA: Allyn and Bacon, Inc.

Office of the Under Secretary of Defense for Acquisition and Technology, Joint Logistics Commanders, Joint Group on Systems Engineering. (1998). *Practical Software Measurement: A Foundation for Objective Software Measurement*, *Version 3.1a*, *April 17, 1998*, United States: Author.

Padberg, F. (2002). A Discrete Simulation Model for Assessing Software Project Scheduling Policies. *Software Process Improvement and Practice*, *9*, 127-139.

Pizka, M. and Bauer, A. (2004). A Brief Top-Down and Bottom-Up Philosophy on Software Evolution. *7th International Workshop on Principles of Software Evolution*, Kyoto, Japan, 131-136.

Pollice, G., Augustine, L., Lowe, C., and Madhur, J. (2004). *Software Development for Small Teams: A RUP-Centric Approach*. Boston: Addison-Wesley.

Pressman, R. (2005). *Software Engineering: A Practitioner's Approach*. New York: McGraw Hill.

Probasco, L. (2001). The Ten Essentials of RUP. *The Rational Edge*. Retrieved on May 1, 2005 from http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/dec00/TheTenEssentialsofRUPDec00.pdf.

Raffo, D., Harrison, W., and Vandeville, J. (2000). Coordinating Models and Metrics to Manage Software Projects. *Software Process Improvement and Practice*. *5*, 159-168.

Raffo, D., Setamanit, S., and Wakeland, W. (2003). Towards a Software Process Simulation Model of Globally Distributed Software Development Projects. Paper

presented at the meeting of the ProSim, Portland, OR. Retrieved January 27,2006, from http://prosim.pdx.edu/prosim2003/paper/prosim03_setamanit.pdf.

Raffo, D., Vandeville, J., and Martin, R. (1999). Software Process Simulation to achieve higher CMM levels. *Journal of Systems and Software*. *46*(2/3), 163-172.

Ramil, J. and Smith, N. (2002). Qualitative Simulation of Models of Software Evolution. *Software Process Improvement and Practice*. *7*, 95-112.

Scacchi, W. (1999). Experience with software process simulation and modeling. *Journal of Systems and Software*. *46*(2/3), 183-192.

Sawyer, S. and Guinan, P. (1998). Software Development: Processes and performance. *IBM Systems Journal*, *4*(4), 552-569. Retrieved January 26, 2006, from the Academic Search Premier EBSCO Host Research database.

Shamsi, J., Chu, C., and Brockmeyer, M. (2005). Towards Partially Synchronous Overlays: Issues and Challenges. Paper presented at the International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications, Orlando, FL. Retrieved on February 24, 2006 from http://www.cs.wayne.edu/~mab/publications/shamsi_j_partialsynchrony.pdf

Smith, A. (1993). *The Wealth of Nations* (E. Cannan, version). New York: Random House Publishing Group. (Original work published in 1776).

Tischler, H. (2002). *Introduction to Sociology* (7th ed.). USA: Wadsworth Thomson Learning.

Weiss, M. (1999). *Data Structures & Algorithm Analysis in Java*. Reading Massachusetts: Addison Wesley Longman.

Wickenberg, T. and Davidsson, P. (2003). On Multi-Agent Based Simulation of Software Development Processes. Retrieved May 1, 2005 from http://www.ide.bth.se/~pdv/Papers/MABS2002.pdf

APPENDICES

APPENDIX A

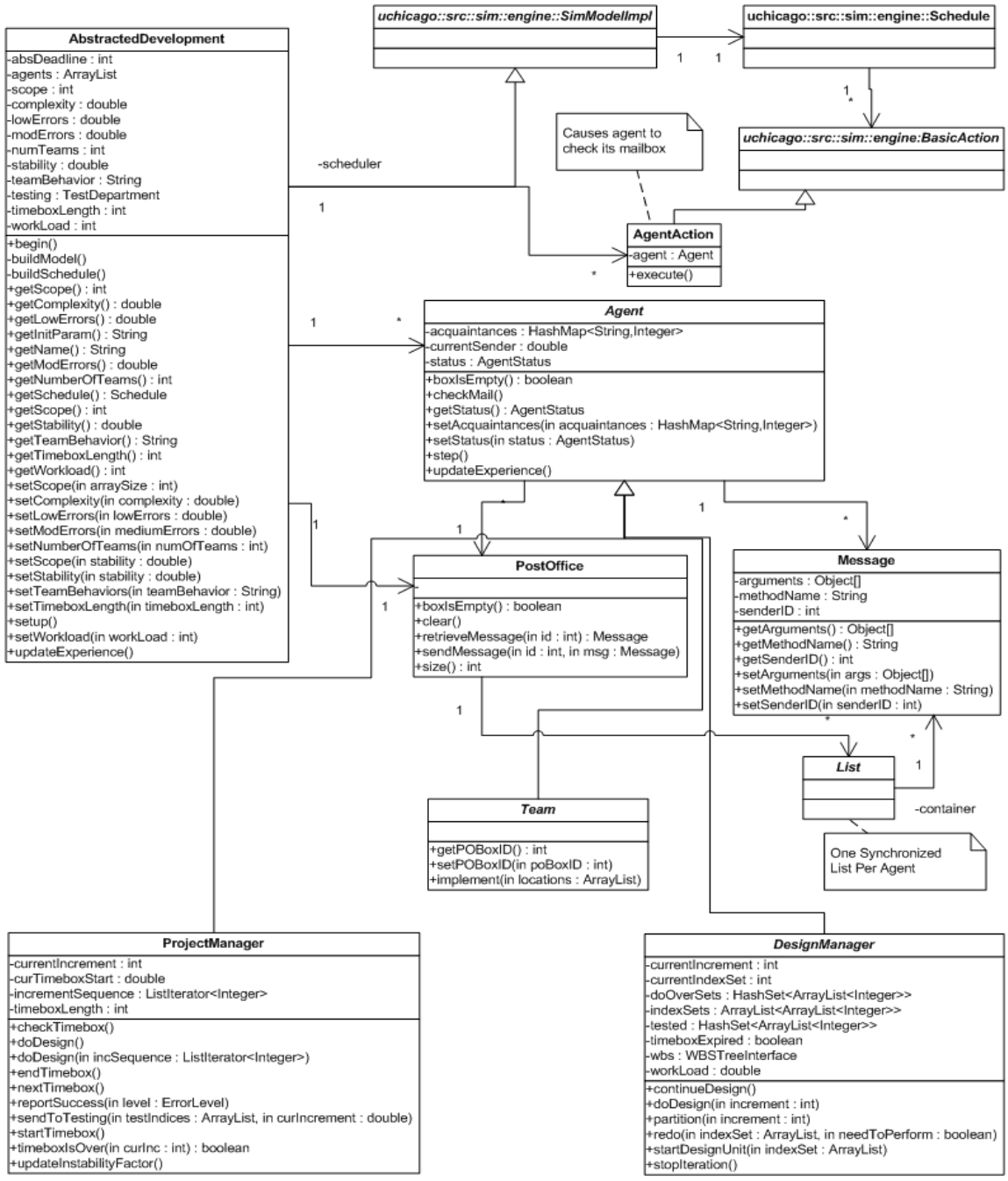

Figure A.1. Team-RUP Design Class Diagram

APPENDIX B

For each experiment in this appendix, we performed 50 replications per replication set. Each statistic in the following tables reflects an arithmetic mean taken across a replication set.

Experiment 1: Agility Test						
REPLICATION SET	*TYPE*	*TURBULENCE*	*WORKLOAD*	*SCOPE*	*COMPLEXITY*	*TEAMS*
1	Agile	Low	5	120	0.492795445	5
2	Agile	Mod. Low	5	120	0.496051976	5
3	Agile	Mod.	5	120	0.49619988	5
4	Agile	Mod. High	5	120	0.500999823	5
5	Agile	High	5	120	0.517297541	5
6	Synchronous	Low	5	120	0.503560578	5
7	Synchronous	Mod. Low	5	120	0.500937619	5
8	Synchronous	Mod.	5	120	0.486810614	5
9	Synchronous	Mod. High	5	120	0.495231466	5
10	Synchronous	High	5	120	0.501508096	5
11	Autonomous	Low	5	120	0.482563196	5
12	Autonomous	Mod. Low	5	120	0.510589997	5
13	Autonomous	Mod.	5	120	0.490344683	5
14	Autonomous	Mod. High	5	120	0.500867129	5
15	Autonomous	High	5	120	0.509257847	5
16	Concurrent	Low	5	120	0.509206642	5
17	Concurrent	Mod. Low	5	120	0.487430728	5
18	Concurrent	Mod.	5	120	0.507232866	5
19	Concurrent	Mod. High	5	120	0.491274695	5
20	Concurrent	High	5	120	0.504179166	5

Table B.1. Factor Levels for the Agility Test

Experiment 1: Agility Test				
REPLICATION SET	PRODUCTIVITY	STAFF UTILIZATION	TIMELINESS	QUALITY
1	0.025328846	0.231140154	0.440316877	0.96509466
2	0.025327223	0.230382858	0.439246148	0.965331789
3	0.025960513	0.230647675	0.435901661	0.965002604
4	0.026268255	0.229222744	0.43890831	0.964103427
5	0.026970226	0.229981082	0.437321071	0.966720034
6	0.023335065	0.642720218	0.292401489	0.071943243
7	0.023615427	0.642125773	0.29284277	0.064871543
8	0.02352542	0.6419996	0.292617552	0.068865499
9	0.024221423	0.641766512	0.290952536	0.072609209
10	0.024997594	0.641311526	0.291381207	0.126947075
11	0.025331712	0.190298548	0.29443539	0.678906763
12	0.026653118	0.190425866	0.252664708	0.687680819
13	0.026146871	0.194835978	0.220097001	0.679297956
14	0.026888272	0.193823064	0.233288766	0.691685123
15	0.027092029	0.19309859	0.209989504	0.701111544
16	0.026411194	0.197775971	0.284803795	0.827527847
17	0.02600541	0.199222974	0.258121465	0.782418654
18	0.026860975	0.200113842	0.235610027	0.768374242
19	0.026630691	0.200132711	0.224455529	0.751175646
20	0.026826836	0.203330749	0.220628672	0.744913224

Table B.2. Response Values for the Agility Test

Experiment 2: Team Size Test					
AGILE					
REPLICATION SET	TURBULENCE	WORKLOAD	SCOPE	COMPLEXITY	TEAMS
1	Low	5	120	0.507566872	2
2	Low	5	120	0.492509003	3
3	Low	5	120	0.494292946	5
4	Low	5	120	0.508440247	7
5	Low	5	120	0.503936501	9
6	Mod. Low	5	120	0.487158775	2
7	Mod. Low	5	120	0.507247658	3
8	Mod. Low	5	120	0.481746178	5
9	Mod. Low	5	120	0.505034409	7
10	Mod. Low	5	120	0.487197266	9
11	Mod.	5	120	0.495859032	2
12	Mod.	5	120	0.492768288	3
13	Mod.	5	120	0.488520584	5
14	Mod.	5	120	0.491357918	7
15	Mod.	5	120	0.507890472	9
16	Mod. High	5	120	0.504902039	2
17	Mod. High	5	120	0.497990532	3
18	Mod. High	5	120	0.510723915	5
19	Mod. High	5	120	0.507437592	7
20	Mod. High	5	120	0.505601082	9
21	High	5	120	0.484216995	2

22	High	5	120	0.499557953	3
23	High	5	120	0.515345497	5
24	High	5	120	0.493893585	7
25	High	5	120	0.483649635	9

SYNCHRONOUS

REPLICATION SET	*TURBULENCE*	*WORKLOAD*	*SCOPE*	*COMPLEXITY*	*TEAMS*
26	Low	5	120	0.506517067	2
27	Low	5	120	0.502209625	3
28	Low	5	120	0.490453186	5
29	Low	5	120	0.504317245	7
30	Low	5	120	0.506039238	9
31	Mod. Low	5	120	0.492678947	2
32	Mod. Low	5	120	0.508490639	3
33	Mod. Low	5	120	0.508122559	5
34	Mod. Low	5	120	0.493064766	7
35	Mod. Low	5	120	0.501874504	9
36	Mod.	5	120	0.498804474	2
37	Mod.	5	120	0.501498833	3
38	Mod.	5	120	0.505621948	5
39	Mod.	5	120	0.501907845	7
40	Mod.	5	120	0.49899334	9
41	Mod. High	5	120	0.509307442	2
42	Mod. High	5	120	0.500957909	3
43	Mod. High	5	120	0.495966949	5
44	Mod. High	5	120	0.513373718	7
45	Mod. High	5	120	0.496787643	9
46	High	5	120	0.50466938	2
47	High	5	120	0.506278419	3
48	High	5	120	0.480222969	5
49	High	5	120	0.512956429	7
50	High	5	120	0.486804581	9

AUTONOMOUS

REPLICATION SET	*TURBULENCE*	*WORKLOAD*	*SCOPE*	*COMPLEXITY*	*TEAMS*
51	Low	5	120	0.504437981	2
52	Low	5	120	0.494516602	3
53	Low	5	120	0.515584564	5
54	Low	5	120	0.500924187	7
55	Low	5	120	0.489090614	9
56	Mod. Low	5	120	0.483288422	2
57	Mod. Low	5	120	0.503692856	3
58	Mod. Low	5	120	0.482916412	5
59	Mod. Low	5	120	0.509558907	7
60	Mod. Low	5	120	0.508256798	9
61	Mod.	5	120	0.501806602	2
62	Mod.	5	120	0.502840004	3
63	Mod.	5	120	0.502829933	5
64	Mod.	5	120	0.496880951	7
65	Mod.	5	120	0.490814857	9
66	Mod. High	5	120	0.490543747	2
67	Mod. High	5	120	0.492832184	3
68	Mod. High	5	120	0.494958954	5
69	Mod. High	5	120	0.501601181	7
70	Mod. High	5	120	0.509886665	9
71	High	5	120	0.504381638	2

72	High	5	120	0.501154861	3
73	High	5	120	0.490907822	5
74	High	5	120	0.4907547	7
75	High	5	120	0.508830681	9

CONCURRENT

REPLICATION SET	*TURBULENCE*	*WORKLOAD*	*SCOPE*	*COMPLEXITY*	*TEAMS*
76	Low	5	120	0.512141724	2
77	Low	5	120	0.50306324	3
78	Low	5	120	0.506592216	5
79	Low	5	120	0.496863937	7
80	Low	5	120	0.493643379	9
81	Mod. Low	5	120	0.505849609	2
82	Mod. Low	5	120	0.503745651	3
83	Mod. Low	5	120	0.509353104	5
84	Mod. Low	5	120	0.508011055	7
85	Mod. Low	5	120	0.487565842	9
86	Mod.	5	120	0.50994812	2
87	Mod.	5	120	0.496735764	3
88	Mod.	5	120	0.498604012	5
89	Mod.	5	120	0.510730362	7
90	Mod.	5	120	0.501268578	9
91	Mod. High	5	120	0.500176506	2
92	Mod. High	5	120	0.51145195	3
93	Mod. High	5	120	0.494881821	5
94	Mod. High	5	120	0.490365829	7
95	Mod. High	5	120	0.506875153	9
96	High	5	120	0.487911987	2
97	High	5	120	0.484253769	3
98	High	5	120	0.506429214	5
99	High	5	120	0.499536057	7
100	High	5	120	0.500761528	9

Table B.3. Factor Levels for the Team Size Test

Experiment 2: Team Size Test

AGILE

REPLICATION SET	*PRODUCTIVITY*	*STAFF UTILIZATION*	*TIMELINESS*	*QUALITY*
1	0.044051743	0.378005524	0.595186005	0.966042023
2	0.034348104	0.312981548	0.521366425	0.966084976
3	0.025159695	0.23119257	0.441159821	0.966228104
4	0.020402493	0.182635918	0.388178291	0.964067993
5	0.016710329	0.151353331	0.348864555	0.959812698
6	0.043163648	0.376634941	0.593599129	0.96248848
7	0.035920982	0.31204361	0.518868103	0.966102142
8	0.024949787	0.231146088	0.443113861	0.964397583
9	0.020536547	0.181842976	0.388081017	0.964103317
10	0.016491178	0.151069822	0.348748589	0.95980011
11	0.043713193	0.376317863	0.596150208	0.958249512
12	0.035134928	0.311978054	0.522726021	0.963125763
13	0.025388844	0.230444336	0.440440102	0.964057159
14	0.02028404	0.181699028	0.387325783	0.960935059
15	0.016987174	0.151326818	0.349285851	0.959202576
16	0.045265956	0.376638374	0.597470741	0.952550964

17	0.035864072	0.311935253	0.52032711	0.96167099
18	0.02662405	0.229608994	0.437987709	0.965223618
19	0.020830314	0.181645393	0.388093567	0.963506622
20	0.016961488	0.150890436	0.348357544	0.95918602
21	0.044276781	0.375775223	0.596898499	0.949793625
22	0.036745334	0.311433983	0.521682777	0.963297119
23	0.026985364	0.229490891	0.437680931	0.965915604
24	0.020424116	0.181579514	0.387562523	0.962319183
25	0.016685095	0.150764418	0.349659653	0.959286804

SYNCHRONOUS

REPLICATION SET	*PRODUCTIVITY*	*STAFF UTILIZATION*	*TIMELINESS*	*QUALITY*
26	0.041179385	0.634200935	0.3962463	0.197500477
27	0.032430801	0.68140007	0.326033211	0.041608076
28	0.023040829	0.642899094	0.294361191	0.035669727
29	0.018192199	0.576714935	0.286881409	0.100382853
30	0.015213822	0.516616898	0.283368053	0.143276119
31	0.041352606	0.633155746	0.390564919	0.149114256
32	0.033630962	0.681340179	0.3247649	0.028966477
33	0.024397221	0.64187973	0.292392254	0.048919411
34	0.018153853	0.576359749	0.287161369	0.059004359
35	0.015038776	0.516223717	0.284099026	0.126769342
36	0.042171841	0.632630692	0.388496323	0.123129187
37	0.034000595	0.679490128	0.321274376	0.067042351
38	0.024155076	0.642472687	0.292336617	0.071619201
39	0.018669291	0.576135025	0.286163197	0.093624048
40	0.015380079	0.515999374	0.282573071	0.13342123
41	0.042980351	0.632119293	0.384306183	0.146244364
42	0.034026537	0.679242325	0.319381485	0.019472237
43	0.024491615	0.642160263	0.291736794	0.070431833
44	0.019296339	0.576832466	0.285464745	0.126118917
45	0.015384932	0.516050453	0.282969418	0.137346582
46	0.043151126	0.631100273	0.381346207	0.206139374
47	0.034948885	0.679811554	0.317774506	0.120081911
48	0.024560139	0.642046814	0.290286999	0.064396501
49	0.019531357	0.575749474	0.285720787	0.15351428
50	0.015693961	0.515414772	0.282995567	0.160785713

AUTONOMOUS

REPLICATION SET	*PRODUCTIVITY*	*STAFF UTILIZATION*	*TIMELINESS*	*QUALITY*
51	0.04633482	0.319705906	0.217772541	0.559238167
52	0.036211948	0.265480137	0.250014458	0.698878021
53	0.026568155	0.19239954	0.281757374	0.741624527
54	0.020516992	0.152884817	0.292305031	0.757556
55	0.016242467	0.122460995	0.316630974	0.803800735
56	0.044749265	0.321155396	0.196761074	0.474672775
57	0.03715286	0.269266834	0.221647549	0.656551666
58	0.025662475	0.193089981	0.251653881	0.670110474
59	0.020459924	0.152909594	0.257929249	0.727322922
60	0.016896591	0.124730186	0.274538136	0.755026016
61	0.04687808	0.326842995	0.160072079	0.445630112
62	0.037131999	0.265239277	0.21048193	0.625786934
63	0.02669383	0.190632534	0.243129787	0.690319214
64	0.020778284	0.156069651	0.236034298	0.721896362
65	0.016755232	0.125230532	0.268688316	0.765222549
66	0.046258535	0.329226723	0.159258509	0.444599686

67	0.037022779	0.264811935	0.210744972	0.640887756
68	0.026191192	0.19473917	0.232602406	0.646861191
69	0.020595665	0.153961849	0.243648319	0.737329788
70	0.017135768	0.12746726	0.24778944	0.752881317
71	0.04684164	0.344193611	0.143241291	0.46414505
72	0.037874436	0.263098373	0.20477314	0.632759666
73	0.02649564	0.199744415	0.215062542	0.653595657
74	0.020625527	0.161586571	0.201779995	0.653775711
75	0.01709568	0.126785088	0.254164219	0.766575089

CONCURRENT

REPLICATION SET	*PRODUCTIVITY*	*STAFF UTILIZATION*	*TIMELINESS*	*QUALITY*
76	0.04670042	0.326758041	0.243448563	0.807180634
77	0.036557169	0.268515205	0.260863037	0.822886963
78	0.026532564	0.19576252	0.28093338	0.834212189
79	0.020177579	0.153982468	0.299499683	0.858339996
80	0.016220839	0.127072706	0.306020412	0.864841003
81	0.046279535	0.32998909	0.214358006	0.752946777
82	0.037313282	0.271869164	0.223312569	0.762961426
83	0.026680987	0.195205994	0.26438303	0.831045456
84	0.020652111	0.156262388	0.274119015	0.836172028
85	0.016284063	0.129494791	0.271660442	0.806089172
86	0.046617651	0.340140877	0.171421528	0.653751526
87	0.036538477	0.271179352	0.206061001	0.729413528
88	0.026788754	0.2007967	0.237194462	0.779173203
89	0.020955527	0.156121264	0.248999977	0.80977417
90	0.016939917	0.129288912	0.259693584	0.81110199
91	0.046227717	0.338670349	0.164122467	0.657293777
92	0.03787406	0.269842796	0.207387581	0.731854095
93	0.026636744	0.201142235	0.221683102	0.767559738
94	0.020457425	0.158801594	0.241037312	0.777952499
95	0.016984755	0.130797281	0.251054211	0.786829834
96	0.046388512	0.346093903	0.152824764	0.6084198
97	0.036829069	0.271491909	0.207048168	0.738629761
98	0.027304196	0.208163147	0.194357815	0.712818146
99	0.021035635	0.158337145	0.234183197	0.775438461
100	0.016927701	0.1298458	0.254258995	0.802528763

Table B.4. Response Values for the Team Size Test