The Dynamics of Capability Development and Erosion

Hazhir Rahmandad
Assistant Professor,Industrial and Systems Engineering
Virginia Tech,
Northern Virginia Center- Rm 430
7054 Haycock Road, Falls Church, VA 22043
703-538-8434
hazhir@vt.edu

Nelson Repenning
Associate Professor, Sloan School of Management, M.I.T
E53-335, 30 Wadsworth Ave.,
Cambridge, MA 02142
617-258-6889; nelson@mit.edu

Abstract

While the notion of a capability is widely invoked to explain differences in organizational performance, we know little about where capabilities come from or why some organizations accumulate them more rapidly than others. To develop an improved understanding of the micro-foundations of capability, we study two software development organizations that showed significantly different capability trajectories despite similar organizational and technological settings. Our analysis of these data suggest the existence of an organizational dynamic that we call the adaptation trap: due to a failure to account fully for delays in receiving information feedback, well-intentioned efforts by managers to search locally for the optimal resource/workload balance lead them to systematically overload their organizations and, thereby, cause capability to erode. We provide a formal model of our emerging theory, the analysis of which provides both insight into the sources of differential organizational performance and testable hypotheses.

Acknowledgments:
Special thanks to Martin Devos, David Weiss, Randy Hackbarth, John Palframan, and Audris Mockus and several members of Sigma organization for their support of, and contribution to, this research. We are also grateful for the valuable feedback of Michael Cusumano, Jan Rivkin, John Sterman, Jeroen Struben, Lourdes Sosa, Gokhan Dogan, and several seminar participants at MIT, Stanford, Wharton, and Academy of Management.

Introduction

The field of strategy has long concerned itself with understanding why some firms do better than others. While arguments focused on position (Porter 1998) continue to offer both significant explanatory power and insight to practitioners, those firms that manage to produce above-average results without the benefit of an obvious positional advantage have increasingly become the focus of scholarly attention. Inspired by the seminal work of Nelson and Winter (1982), Wernerfelt (1984), and Barney (1991), a growing number of strategy and organizational scholars now look inside the firm for the sources of sustained competitive advantage and performance heterogeneity.

As befits the diversity of the perspectives that characterize management and organization theory, the search for competitive advantage within the firm has been guided by several theoretical and methodological frames. Scholars favoring large-sample approaches have produced several analyses suggesting that, in at least some industries, heterogeneity in firm performance cannot be fully explained by positional differences (Rumelt 1991; Mauri and Michaels 1998; Spanos and Lioukas 2001). This line of work suggests both that what happens inside the firm matters and that such differences in internal operations are not quickly eliminated by imitation (Henderson and Cockburn 1994; Roberts and Dowling 2002). Building on these large-sample observations, scholars with a more theoretical bent have pursued a number of paths towards a convincing explanation of the observed differences in performance. The literature now includes both a growing collection of formal models that demonstrate the difficulties associated with imitating successful firms (e.g. Rivkin 2001) and a set of studies detailing both the development of internal

resources and assets (Siggelkow 2002) and, most notably, the failure of firms to adapt those resources to changing environmental requirements (e.g. Henderson and Clark 1990).

The central notion arising from this line of inquiry is that of the *capability*, roughly defined as a set of routines that enable a firm to produce a particular output given a set of inputs and targets (e.g., develop new products) (Winter 2003). A capability, the current view holds, differs from the normal conception of an asset in that it cannot be purchased directly but instead is the accumulated product of a particular set of activities (e.g. Dierickx and Cool 1989; Winter 2000). Positing capability as the product of accumulation creates several platforms from which to build a theory of persistent heterogeneity in firm performance. Most fundamentally, capabilities in this conception cannot be acquired in external markets, thus eliminating one of the most powerful forces for competitive equilibration. Instead, firms wishing to close a competitive gap must resort to either imitation—growing their own capabilities to match those of a competitor—or substitution (Dierickx and Cool 1989; Peteraf 1993).

Both paths to strategic parity are, however, fraught with pitfalls. Imitating the processes through which a particular firm developed a given capability is impeded by the sheer complexity of the modern organization (Rivkin 2000; Rivkin 2001). For example, despite having been the subject of scholarly attention for more than two decades, the question of which elements of the Toyota production system are responsible for Toyota's manufacturing capability remain much debated and few substitutes have emerged for Toyota's unique abilities. By positing the existence of a transformation process that cannot be purchased directly, the notion of capability provides a

platform from which to resolve one of the key questions facing organizational scholars: why do seemingly similar firms produce significantly different levels of performance?

Despite its promise, however, the explanatory power of theory built on the existence of capabilities remains modest, and the lack of progress can be traced to a limited understanding of their origins (Helfat 2000; Helfat and Peteraf 2003). We know little about the specific mechanisms through which capabilities are created and sustained and have no theory detailing the actions that firms take to foster their accumulation. Building on Nelson and Winter (1982), several theorists have advanced routine-based views in which capabilities are said to result from the repeated execution of specific routines and learning (Zollo and Winter 2002). Given the breadth of organizational action that stems from routines, however, this view offers little analytical purchase; current theory offers few criteria on which to judge whether a given set of activities and learning mechanisms yield a particular capability. Consequently, the current literature lacks a general explanation for why some firms accumulate capability faster than others. Until such a theory exists, the capability concept offers little utility in understanding the sources of heterogeneous firm performance (Williamson 1999; Priem and Butler 2001).

Capability Dynamics and Dynamic Capabilities

The lack of the useful theory concerning the origin of capabilities can be traced to three sources, conceptual ambiguity, operational distance, and research design.

Conceptual Ambiguity. The discourse on capabilities lacks conceptual precision on a key front. Recently, scholars have shown significant interest in the notion of a dynamic capability—

defined as a routine or skill that modifies an operational or "zero level" capability. The recent focus on dynamic capability appears to have obscured the fact that, except in a few special cases, capability is itself a fundamentally dynamic notion. Early critics of the resource-based view of the firm challenged its utility in explaining differential heterogeneous performance by suggesting that resources central to firm performance could, in principle, be purchased in external markets, thus eliminating them as a source of differential competitive advantage. Following the conceptualization of Dierixx and Cool (1989), scholars responded by positing strategically important resources, soon to be called capabilities, as entities that, by definition, could not be purchased, but could only be accumulated through doing particular sets of activities within the firm. Such accumulation processes could, this view held, result in the ability to execute certain transformation processes in ways that would be difficult for other firms to duplicate.

Subsequent scholars labeled these capabilities, those focused on transforming inputs into outputs, as zero-level or operational capabilities, reserving the label "dynamic" for the ability to modify operational capabilities (Teece, Pisano et al. 1997; Eisenhardt and Martin 2000; Winter 2003). The result is that with a few notable exceptions (e.g. Helfat and Peteraf 2003), operational capabilities have been treated as entities that can be reconfigured quickly through dynamic capabilities (Schreyogg and Kliesch-Eberl 2007). Such a framing, however, directly contradicts the conceptualization of capabilities as the product of accumulation since accumulation is, by definition, a dynamic process (Sterman 2000) and major shifts in capabilities appear to require significant periods of time (Argote and Epple 1990; Sterman, Repenning et al. 1997; Repenning and Sterman 2002). Thus, a conceptualization of operational capabilities capable of explaining firm heterogeneity must account for their fundamentally dynamic character. While the available

evidence suggests that capabilities can, with the passage of time, be both gained and lost, existing theory offers little to explain how or why.

Operational Distance. The second limitation of existing theory lies in its distance from the underlying phenomenon. As argued by several scholars, the study of, and theory about, organizations has become increasingly removed from the day-to-day activities that constitute organizational life (e.g. Barley and Kunda 2001; Kaplan and Henderson 2005). Such a critique is particularly relevant to the study of capability. While some theoretical frameworks capture the learning processes underlying capability evolution (Zollo and Winter 2002; Gavetti 2005), empirical work establishing the micro-foundations of such frameworks is virtually non-existent. Capabilities are not instantiated in the boardroom, but instead reside on production lines, in R&D labs and at the point of customer contact. Few scholars have, however, systematically observed the locations at which capabilities are said to reside.

Such operational distance poses a severe limit on the continued growth of the understanding of capabilities and their connections to firm-level heterogeneity. Without a clear mapping between theoretical concepts and concrete observations, scholars will be hard pressed to systematically confront theories premised on capabilities with data. Worse, without a clearly specified set of mechanisms grounded in observable features of organizational life, empirical research focused on capabilities runs a significant risk of being tautological; capabilities are those things which good firms have rather than being the true source of above-average performance (Williamson 1999; Priem and Butler 2001).

Research Design. Many of the limitations of current theory on capabilities can be traced to the design of existing research. Of the few studies that attempt to observe capabilities in action, most focus on how firms react to radical technological change (e.g. Henderson and Clark 1990; Rosenbloom 2000; Tripsas and Gavetti 2000; Keil 2004) (See (Raff 2000) for a notable exception) . Such a research strategy risks conflating two conceptually distinct determinants of firm performance, the firm's ability to react to change and the firm's ability to execute a distinct set of policies to produce capabilities. While strategy scholars have made substantial progress in understanding how existing capabilities constrain the ability of firms to react to changes in their competitive environment (Leonard-Barton 1992), we know relatively little about how those capabilities got there in the first place (Gavetti 2005; Lavie 2006). Moreover, while scholars have recognized that the current level of capability is determined by both creation and erosion processes (Helfat and Peteraf 2003), we know little about the micro-processes underlying the erosion of capabilities. Thus, the research designs that have dominated the study of capabilities have created a significant blindspot in our understanding of the sources of competitive advantage. Until we understand the micro-processes through which capabilities are built, maintained, and lost, scholars will be ill-positioned to explain differences in performance that do not stem primarily from an advantaged market position.

Studying the Micro-Foundations of Capabilities

Studying the origin of capabilities poses several significant challenges. First, capabilities are embodied in the adherence to, and departures from, standard procedures, and are often not things that can be counted, thus making it difficult to identify those activities that contribute to competitive advantage. As much research in strategy and organization has demonstrated, such

situations are prone to spurious causal inferences and consequent superstitious learning—a situation that afflicts researchers and practitioners equally. Three research design elements are necessary to offset this challenge. First, following the discussion above, capabilities, even operational ones, must be studied longitudinally. If, capabilities are accumulated through a particular set of activities, then their origins can only be understood by explicitly focusing on the accumulation and erosion processes.

Second, a comparative design is essential to isolate those practices that actually differ across organizations that display significantly different levels of performance. Cross-sectional comparison, while helpful, is not sufficient to eliminate the potential for causal ambiguity as higher performing firms may differentially adopt irrelevant practices. Thus, the third necessary element is a clearly specified and testable causal theory that yields inferences connecting more primitive features of the organization in question to the capabilities that are hypothesized to generate differences in performance outcomes.

Short of a true randomized experiment, the issue of causality can only be sorted out by developing a process-based theory that links structural variables outside the firm's control to the creation and sustainability of a given capability. Such a theory would allow analysts to treat differences in industry structure and technology as a natural experiment, the analysis of which might yield testable predictions concerning the types of capability that are both valuable and hard to build and sustain within a specific industry structure. For example, a theory that indicated that senior executive team flexibility was a relatively more important and harder to sustain in rapidly changing markets would then predict that the return to flexibility would be relatively larger in

such markets than in those with a slower rate of change. A study designed to *develop* new theory cannot, on its own, eliminate the chance of spurious inference (since it is by definition sampling on the dependent variable). But, by specifying a logically consistent theory, such a design enables the derivation of testable hypotheses and thus, ultimately, provides a check on its utility.

The necessity of a comparative design creates an additional challenge because the level of capability and its rate of accumulation are influenced by innumerable variables beyond the practices that instantiate them. The relative importance of a given capability varies across industries—reliability is more important in airframe construction than in word processing software— and thus firms are likely to focus on them differently. Focusing within a specific industry segment and technology reduces this problem, yet firms within a given industry may have different positions and, therefore, face different constraints in their ability to invest in different capabilities, thus creating an additional source of variability.

The ideal research design would thus allow one to compare the accumulation paths of an important capability across two different organizations that displayed significant differences in performance while holding constant as many outside variable as was possible. From such a comparison one could then induce a theory detailing the practices that contribute to the successful accumulation and maintenance of one or more important organizational capabilities. While no actual design can exactly match this ideal, the company discussed in this paper presented us with the opportunity to study two organizations engaged in the same basic task, software development, facing virtually identical resource constraints, but who nonetheless displayed significantly different trajectories in their core capability. Comparing these two

organizations and their approach to software development yielded a theory of capability development, maintenance and erosion with sufficient specificity that it can be formalized in a mathematical model. The resulting characterization of capability dynamics yields both new insights into the reasons that some firms accumulate capabilities while other do not, and testable hypotheses.

DATA AND METHODS

The Research Site

Sigma is a large telecommunication firm that develops dozens of different software and hardware products. Approximately 16,000 people work for Sigma and are distributed in multiple locations around the world. In the last five years the company has shifted its strategy towards the software side of the business. Given this emphasis, efficiently developing high quality software is core to Sigma's ongoing success—it is an important operational capability. Despite its centrality to company's mission, however, there is substantial variation in software practice across Sigma's divisions. Some areas consistently produce high quality code while others are plagued with low productivity and high defect rates. Not surprisingly, this variance is of great concern to Sigma's leadership and they were thus eager to participate in a study of their development efforts.

To isolate the practices that separated high performing areas from low performing ones, and thus hone in on the micro-foundations of a specific capability, we chose a polar types research design (Eisenhardt 1989). With Sigma's help, we identified Alpha and Beta, two products that showed significant differences in performance outcomes despite comparable technical complexity,

resource loading, and incentive structures. Both are large software systems with multiple releases, follow similar official development processes and have similar incentive structures. Despite these similarities, and the initially successful market performance of both products, the processes used by the two development teams have diverged over time. Beta's practice continues to follow Sigma's suggested practice closely. Alpha's practice, in contrast, has changed significantly, moving progressively towards a "firefighting" mode in which its development resources are increasingly focused on fixing problems rather than the activities that would have prevented those problems.

The two projects also displayed significantly different performances in terms of quality, timeliness, and productivity. The Beta team had a history of releasing high quality software on time with few bugs and customer complaints. Alpha's initial releases were also highly successful, but its subsequent performance deteriorated significantly. The comparison of these two projects thus provided us with an opportunity to study how organizational capabilities evolve over time and can converge to markedly different states despite similar exogenous conditions and the absence of technological shocks. Comparing and contrasting what happened in the two cases provided a rich context to build a model linking specific practices to the ongoing evolution of an important capability.

Data Collection

The data collection began with the first author spending three months on site at Sigma. During that time he conducted 48 initial semi-structured interviews, each of which was done either in person or on the phone and lasted between 30 and 90 minutes. The initial interviews included members of all functional areas involved in development, services, and sales, architects and

systems engineers, developers, testers, customer service staff, sales support personnel, and marketing personnel, as well as managers up to two layers in several of these areas. During the initial three month period, the first author also attended group meetings focused on the progress of the products under study and conducted an extensive review of the available archival data. Later, as the key themes emerged from the analysis, additional interviews were conducted to evaluate emerging hypotheses and to fill in gaps. These efforts included 22 additional interviews and a group session that elicited ideas from 26 experienced members of the organization. The first author also visited two other sites where work on Alpha and Beta was being conducted. Additional data were also collected as we began to refine the main focus on our theorizing.

The bulk of the interviews were focused on understanding the detailed processes that individuals and groups used in their development activities. The actual processes used to develop software (as opposed to that captured in the Sigma manual) is composed of the routines and operating procedures used on a day-to-day basis. Much of our effort focused on uncovering this process and clearly separating what was actually done from what Sigma prescribed. The archival data, mostly from Alpha's case, included project planning and review files for different releases of the product, internal project assessments conducted at the end of major projects, and quantitative data on code development and bug fixing, human resources, and problems reported by customers.

Analysis

We analyzed the data in two overlapping phases. The first phase followed standard practice for qualitative research (e.g. Glaser and Strauss 1967) and focused on identifying emerging categories and connecting causal hypotheses. In the second phase we integrated these

hypotheses into a single cohesive theory via the development of a formal model. Although we describe these two phases in sequence, they overlapped significantly. Attempting to capture our observations in a formal model often necessitated a return to the source data to answer questions raised by the modeling that had not been part of the initial hypothesis generation phase.

Qualitative Hypothesis Generation. Interviews were recorded and a summary file of each interview was created soon after the interview session. In each interview we searched for factors connected to the practice of developing software and how those factors might relate to major performance metrics (quality and schedule in this case) and looked for corroborations or disagreements with old factors. Based on these factors and the observations on the site, we generated several different hypotheses about the processes that contribute to the observed differences in the capability levels and performance outcomes across multiple releases of Alpha and Beta. The complete set of these dynamic hypotheses is documented in the online technical appendix. We used this information to build a conceptual model of how the mechanics of the development process work. We then narrowed the list of hypotheses into a core set based on the themes that were most salient in the interviews and in the history of the products.

Formal Theory Development. The set of dynamic hypotheses that emerged from our data constituted a qualitative theory explaining the differing performances of Alpha and Beta. While such theories are widely offered in the organizational literature, a variety of studies show that our ability to conceptualize and draw correct inferences from even the simplest dynamic mechanism is quite limited (See Sterman 1994 for a summary). To offset these limitations, we used our qualitative analysis as the basis for constructing a formal model to explain the dynamics we observed (Forrester 1961; Sterman 2000). We used archival data and the judgment of

organizational participants to parameterize the model. The modeling process included first building a detailed model validated by the historical performance data and then extracting the main dynamics in the form of a simpler, more stylized model that captures the essential elements of our emerging theory. This simple, generic model is discussed in detail in this paper (a full report of the detailed model can be found in Rahmandad 2005).

The formal model served two purposes in our effort to identify and understand the sources of capability. First, by flagging both missing and inconsistent components in our initial set of hypotheses, the formal model enforced the internal consistency of our theory. Iteratively moving between model and data highlighted the pressures and misperceptions that led to the persistence of different decision-rules in the two projects. These decision-rules combine with the physical constraints inherent in developing software to explain the differential accumulation of development capability between Alpha and Beta and their different performance levels. Second, via sensitivity analysis and other simulation experiments, the model enables the derivation of sharp and testable predictions from our theory. Most notably, our predictions extend beyond statements that link capabilities to outcomes, and identify the contextual variables that determine the relative value of the capabilities we study. While not completely eliminating the possibility of spurious inference, such specificity positions future researchers to provide more compelling empirical tests of the existence and contribution of capabilities hypothesized to create competitive advantage.

VARIANCE IN SOFTWARE DEVELOPMENT AT SIGMA

The Espoused Development Process at Sigma

We first outline Sigma's prescribed software development process, which all projects are supposed to follow, and then elaborate on the differing performance of the two projects we studied. The practices that constitute an effective software development process are now well-established (Jones 2000), and Sigma's proscribed process followed accepted best practice for waterfall development closely (waterfall or serial development is typically best for large and well-defined software projects). Despite this acceptance, many projects at Sigma, including Alpha, found it difficult to follow these practices on a regular basis. Thus, the capability at issue is more than *knowing* the right things to do, but also includes the ability to embody those practices on a day-to-day basis.

Software is typically developed in multiple releases, each with three main phases, which can be followed serially (i.e. waterfall development, typically for larger and well-defined projects) or iteratively (for smaller projects or those with unclear requirements). During the period under study, Sigma's development process was largely serial with the occasional iterative element. In the concept design phase, the main features to be added to the next release are chosen, the product architecture is designed, and general requirements for developing different pieces of code are established. For example, a new feature for the next release of a call-center system could be "linking to national database of households and retrieving customer information as the call is routed to a customer service agent." Software architects determine the method by which this new feature should be incorporated into the current code, which new modules should be written, and how the new code should interact with the product. Subsequently, a more detailed

outline of the requirements is developed that describes the inputs, outputs, and performance requirements for the new modules, as well as modifications of the old code.

The next step of the software development process, usually the largest share of the work, is developing the code. Software engineers (developers) develop the new code based on the requirements they have received from the concept design phase. The quality of the development depends on several factors, including the skills of individual developers, complexity of the software, adherence to good development practices (e.g., writing detailed requirements and doing code reviews), quality of the code they are building on, and quality of the software architecture (MacCormack, Kemerer et al. 2003). Preliminary tests are often run in this stage to insure the quality of the new code in isolation before it is integrated with other pieces and sent to the quality assurance stage.

Quality assurance entails running different tests on the code to find problems that stop the software from functioning as desired. Often these tests reveal problems in the code that are referred back to developers for rework. This rework cycle continues until the software passes most of the tests and the product can be released. Note, however, that not all possible errors are discovered during testing. Often tests cover only a fraction of possible combinations of activities for which the software may be used in the field. Therefore, there are always some bugs in the released software.

When the software is released, customers (typically large organizations in Sigma's case) buy and install it and subsequently require service from the company. The service department follows up

on bugs reported by customers and refers these bugs to the development organization. Bugs, if they significantly detract from customer satisfaction, are fixed at the customer site through patches and ad hoc fixes, an activity known as *Current Engineering* (CE). CE is often done by the same team that initially developed the code. If the bug is not critical, then it can be fixed in subsequent releases of the software, an activity known as *Bug Fixing*. Bug-fixing usually competes with addition of new features to the next release.

The development, launch, and service activities discussed above focus on a single release (version) of the software, but there are typically multiple releases in progress at any given point in time, resulting in two key interconnections across different releases in progress. First, subsequent generations of software build on their predecessors and hence carry over the problems in code and architecture if these problems are not remedied through bug-fixes or refactoring of the architecture (*Refactoring* is improving the design and architecture of the existing code.) Second, because they all draw off a common resource pool of developers, resources must be shared among new development, refactoring, bug-fixing and CE. The quality of past releases, because it determines the level of CE and bug-fixing, thus has a significant influence on the resources available for the development of new features in the current release.

Project Beta

Product Beta is part of a complicated telecommunication switch that is developed largely independent of the parent product but is tested, launched, and sold together with the parent product. While Beta includes both software and hardware elements, the majority of its over-80-person R&D resources focus on developing the software. Beta's R&D team is spread across multiple locations with four main sites, two of which overlap with Alpha.

The Beta organization follows the proscribed Sigma practice closely. Its development practice includes early review and testing, root-cause analysis, and extensive automation of testing. The specificity of requirement documents along with comprehensive review in the coding steps enable the avoidance or early discovery of problems. Moreover, early testing of code inside the development organization is prevalent in Beta, resulting in a small number of bugs once the product is sent to the quality assurance organization for integration and performance testing. Field problems are sufficiently infrequent that when one does appear, the development organization can often do a root-cause analysis to determine its sources, thus identifying and eliminating a potential failure mode in the underlying process.

Beta has had exceptionally good quality and timely delivery through its life. The product is recognized throughout the organization for its high quality and consistent delivery. While in other products it is not unusual for thirty percent of the development resources to be engaged in current engineering activities, at Beta this number has always remained under ten percent. Beta has also delivered most of its releases on time. In two cases Beta had to delay a release to remain in synch with its parent product, which was delayed. Beta's quality and size are reported in **Figure 1**.

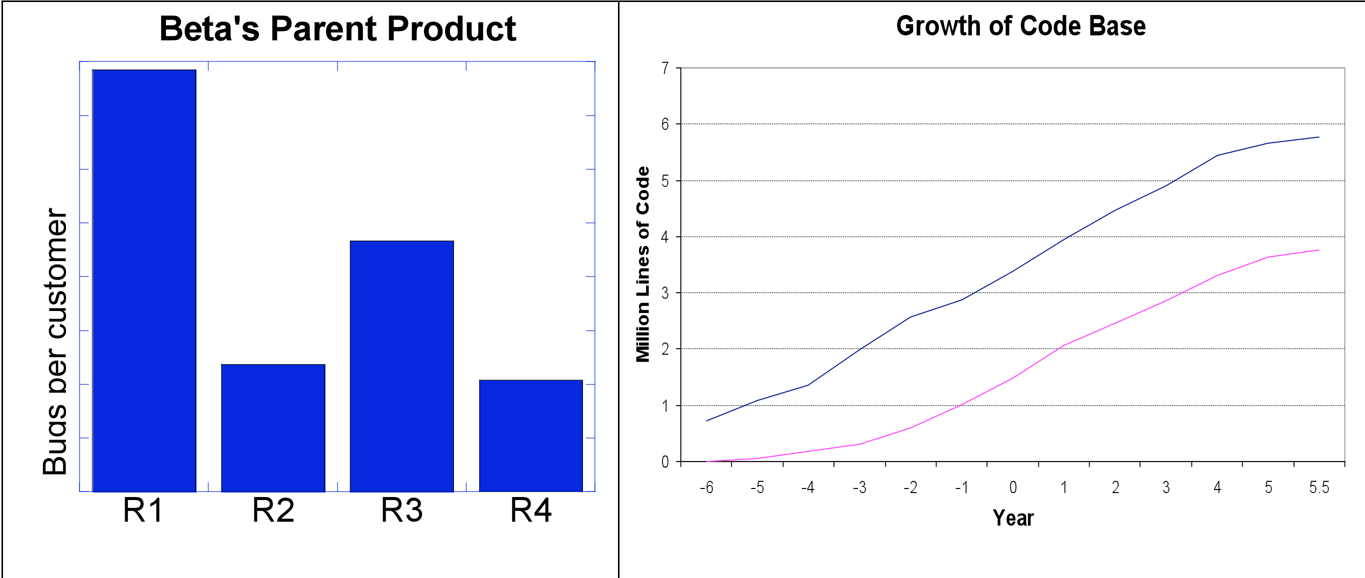


Figure 1- Beta's (parent product) quality (left) and size (right). Quality is measured in bugs reported per customer in the three months after the installation of a release. Scale is removed for confidentiality. Size is reported in new lines of code added to the code base, and year zero is when Alpha joined Sigma organization.

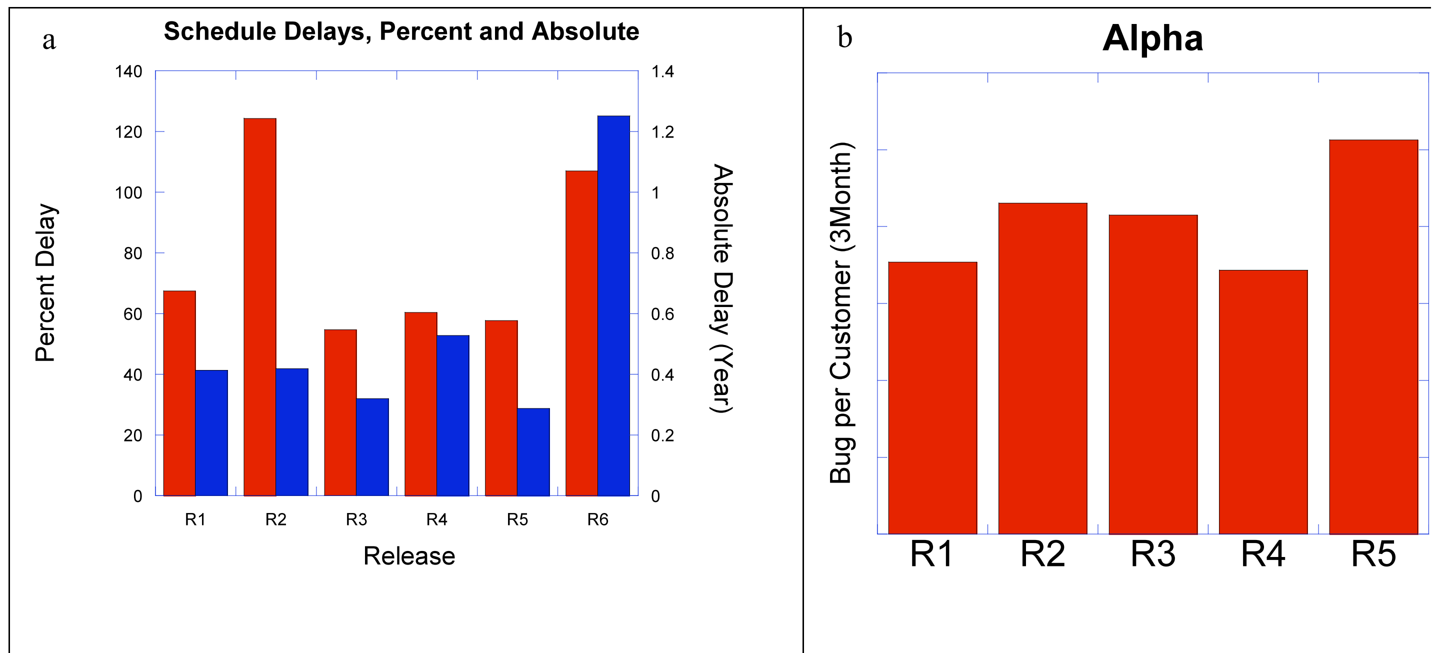
Alpha

Product Alpha is a Customer Relationship Management (CRM) software system used to manage call centers. Alpha was initially produced by a small start-up firm that was then acquired by Sigma. Alpha has been part of Sigma for 8 years and the team working on it has fluctuated around 120 full-time employees who work in five main locations. We focused our data-gathering effort on the 4 recent years of its history. Its initial releases met with significant success, and subsequently led the market; at the time Sigma considered Alpha to be a strategically important and promising product. However, during the last two years we studied, due to long delays in delivery and low quality, Alpha ceded its leadership position. The decline has been so significant that its long-term viability is in question.

In contrast to Beta, Alpha's adherence to best practice and performance have eroded with time. The official development process calls for comprehensive requirement review, development of detailed requirements, code review sessions in which individuals review each others' work, and multiple testing and documentation steps throughout the development, but Alpha often departs from this process. And, although it was initially a high quality product, Alpha's developers cope with increasing time pressure by skipping many of the review and testing steps early in the development process. Experienced developers say that the level of requirement review is low and a growing fraction of the development work starts based on aggregate requirements (rather than the detailed requirement specified by the Sigma development process). This practice creates the danger of missing potential interactions among modules of code, errors in the interfaces between different modules, and complications in integration testing. Code review and unit testing have also suffered, leading to a growing chance of problems surfacing later in the testing process and unexpected surprises. Documentation of code is now far from complete and developers have a hard time building on old modules, which are often not well-documented. While reviewing and testing code early in the development process is costly in the short-term, research suggests that these investments yield a favorable return due to the avoided rework that would otherwise be necessary when problems are identified through quality assurance or by customers (Jones 2000). Alpha's current practice no longer includes many of these basic activities and consequently its performance has deteriorated on several important dimensions, as reflected in the comments of an experienced developer:

> *"What I have seen is years and years of degradation – getting worse and worse over time."*

Software development performance is often measured on three dimensions, schedule, quality, and productivity, and Alpha's performance deteriorated in each category. Timeliness measures how well a project has followed its schedule and has a strong impact on profitability (Ethiraj, Kale et al. 2005). Measured either in absolute terms or as a percent of the project's original schedule, Alpha's timeliness worsened or did not improve over the six releases we studied (**Figure 2**-a). Quality is typically defined by customer's experience of the software and can be measured by the number of serious bugs reported per customer per unit of time in the first few months of installation of the software. As shown in **Figure 2**-b and 2-c, the number of bugs in Alpha grew on average over the period of study. Finally, the productivity of the team can be measured in terms of the number of lines of code developed per individual involved with the R&D process. As shown in **Figure 2**-d, Alpha's productivity declined significantly over the second half of the study period.



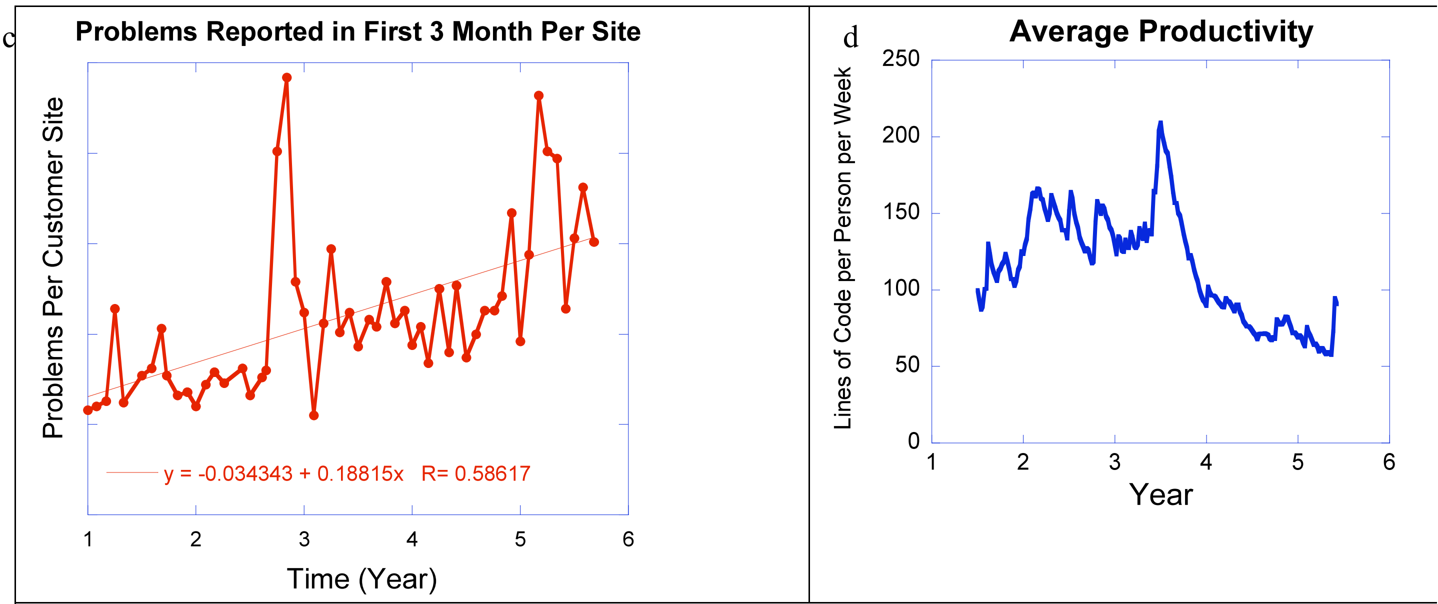


Figure 2- Alpha's performance on schedule (a), quality (b and c), and productivity (d). a) The absolute (right bars) and percent (left bars) of delay for different releases of Alpha. Delays are expressed as the difference between the originally estimated lengths of the projects and their actual lengths and show no improvement pattern. b) The average number of bugs per customer in the first three month of installation, for different releases of Alpha. c) The same measure as b, but shown continuously. The slope of the curve is significantly positive, suggesting a gradual decline in quality. d) The productivity of Alpha team, expressed in the average number of lines of new code per person per week, shows a significant decline after year three.

Research Questions

The histories of Alpha and Beta pose two challenges to the current understanding of organizational capabilities. First, although Alpha was deemed strategically significant and therefore received the continued support of Sigma's leadership, its performance demonstrably declined over the period of study. Given the lack of a significant technological or economic disruption during this time, existing theory offers little to explain why. To the contrary, the current view holds that organizations are fundamentally adaptive (Nelson and Winter 1982), resulting in ongoing learning and improvement (Argote and Epple 1990). While routines are recognized to be susceptible to convergence to local maxima and therefore often impede an organization's response to technological or environmental shifts (Levitt and March 1988;

Leonard-Barton 1992; Levinthal 1997), and absolute decline in the effectiveness of routines in a stable environment is not predicted by current theory.

Second, although Alpha and Beta shared many attributes including, size, resources and organizational and incentive structures (several of the people involved on the two products even worked in neighboring office), their results differed significantly. A researcher in Sigma familiar with both products summarized the state of the two development groups by saying:

> *"Projects [Alpha] and [Beta] are significantly different in their development processes and their results. [Alpha] is a project that has been in continual turmoil, producing much code, often late and of poor quality and not selling well in the market. It must devote considerable resources to fixing field errors. [Beta] … is almost always on time and produces high quality code with few field errors that sells well in the market"*

Due to random changes in the customer demands, personnel turnover, and other exogenous factors, transient differences in efficiency and practices that constitute development capability are expected even across similar organizations. One would not expect these differences to be permanent, however, as neighboring organizations learn from the experience of others and converge to similar routines and performance. In contrast to the current view, Alpha and Beta, despite similar organizational and technical conditions, displayed diverging trajectories of both performance and practice. In the next section, we use this empirical anomaly to theorize about dynamics of capability evolution, specifically addressing the question of how differential performance can emerge in product development capability and why it persists. Based on these two cases we use an inductive approach to build a formal model of the dynamics that govern the evolution of development capability and explore how these dynamics can lead to heterogeneous performance across different organizational units.

Model and Analysis

We develop our answer to these questions in two pieces. First we propose and analyze a model to explain the declining performance of the Alpha project. Building on a simple representation of the "physics" of developing software, we show how the intendedly rational decisions of participants within that process combined to create a phenomenon that we call the *adaptation trap*. The adaptation trap details how the well-intentioned efforts of managers to optimize the balance between the demand for new features and the resources to develop those features can inadvertently trap the project in a vicious cycle of declining productivity and increasing demands. Second, we interpret the Beta project's practice within the context of the adaptation trap and show how Beta's managers successfully avoided trap that proved so costly for Alpha.

The Rise and Fall of Alpha: The Adaptation Trap

We develop the concept of the adaptation trap in three steps. First, we show how the basic physical structure of any work process when coupled with the effects of work pressure on productivity creates a system with a tipping threshold, a level of activity that once exceeded causes a sustained decline in performance. Although it is derived from very different assumptions, this model is structurally similar to that offered in Rudolph and Repenning (2002), thus we present it briefly and refer the reader to the original paper for more details. Second, we extend the model to capture how the dynamics change when the work process in question creates an installed base of products that can require additional attention (e.g., bugs that need to be fixed). Finally, we include the dynamics that emerge when the product (e.g., the software system) has an underlying architecture that extends across multiple generations. These additions

both add to the realism of the model and help explain the persistence of differing performance between Alpha and Beta.

Work Pressure and Tipping. Figure 3 shows the stock and flow structure of our initial model formulation. The stocks (boxes) represent the integration of different flows (thick valves and arrows) and help distinguish between a particular action or activity (represented by a flow) and its accumulated impact (represented by one or more stocks). Stock flow networks constitute the foundation of any dynamic system (Sterman 2000) and are central to characterizing its behavior. The first construct in our model, *Demand for New Features*[1], represents the flow of requests for additions to the system. Feature demands often originate in market research, customer focus groups, benchmarking with competitors, and other strategic sources. In both Alpha and Beta, new features were proposed by sales, services, and marketing groups, and further augmented by the R&D managers. The demand for new features is largely exogenous to the R&D department, yet product managers have some flexibility in prioritizing among potential features (see **Figure 3**).

The flow of newly proposed features accumulates in the stock of *Features Under Development.* Feature requests remain in this stock until they are developed, tested, and released in the market to become part of the *Features in the Current Release* of the software. The flow *Feature Release* captures the speed at which the PD organization develops and releases new features and therefore how often new releases are launched.

[1] The names of variables that appear in the model diagrams are *italicized* when they are first introduced.

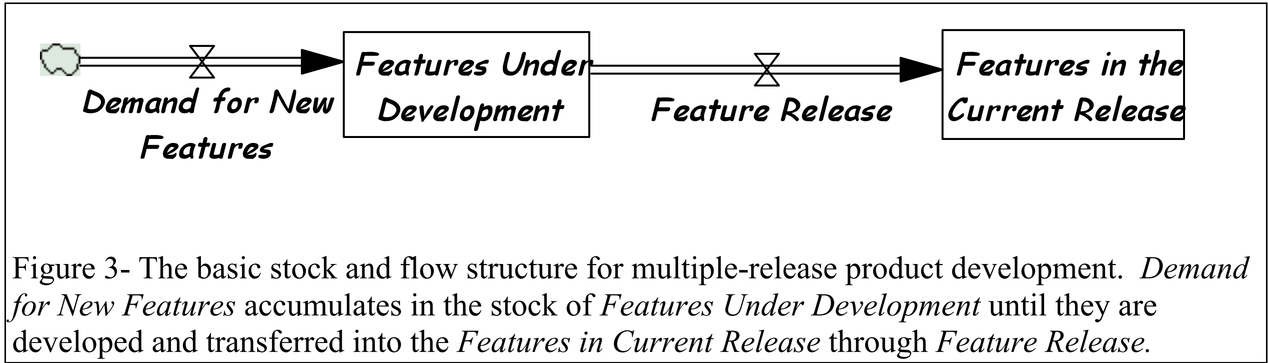


Figure 3- The basic stock and flow structure for multiple-release product development. *Demand for New Features* accumulates in the stock of *Features Under Development* until they are developed and transferred into the *Features in Current Release* through *Feature Release.*

Product development capability manifests in this setting as the ability to transform feature requests into defect-free features that appear in the next release of the product. Companies that can sustain high rates of feature release for a given resource base will have a competitive advantage through some combination of more features, faster release times, higher product quality, and lower product and development cost.

In its simplest form, Feature Release (FR) depends on the amount of resources available for development (R_D), the gross productivity of these resources (P), and the quality of their work (e) (i.e., the fraction of accomplished work that passes the quality criteria). Consequently *Feature Release* can be summarized as:

$$FR=R_D*P*(1-e) \qquad (1)$$

Productivity (P) (e.g. in features per developer per month) depends on several factors, including the skill and experience of individual developers, the quality of design and requirements for the development, the complexity of the code, and the development process used. Similarly, the Error Rate (e) (a fraction between zero and one) depends on multiple factors, including the quality of design and detailed requirements, complexity of architecture, the scope of testing by

developers (before the official testing phase), code reviews and documentation, and the pressure developers face.

It is difficult to attribute Alpha's decline to exogenous factors both because there were few such changes over the study period and, during that time, Beta, which faced similar constraints, flourished. Thus, to understand Alpha's decline we focus on the management of the project. Our data suggest that a key difference between the two projects was the number of pending requests for new features. Beta tended to face fewer such requests and was not willing to adjust its approach to development when the demand for new features exceeded its ability to deliver. Alpha, in contrast, tended to face a larger stock of new requests and, importantly, often attempted to respond to those requests. An Alpha manager elaborates on this difference:

> *"[Beta Group] looks at what are the critical features, what are the extra features, and they pull features out [if unrealistic] We are not good at that. "*

To model the differing approaches to managing the work-resource balance we begin with the notion of the *Resource Ratio (RR),* defined as the ratio of resources required to finish the development of the pending Features Under Development by a scheduled date to the available resources. The Resource Ratio is a function of three related decisions: how many features are under development, how much time is available to develop these features, and how many development resources are available to do the job. The Resource Ratio is a measure of the *pressure* on the development team to deliver their work more rapidly—when the ratio is low, the team can meet its targets relatively easily; when the ratio is high achieving targets is difficult. Pressure is an important determinant of development performance as it can have a significant effect on the speed at which people work (Mandler 1984; Fisher 1986). Abdel Hamid and

Madnick (1991) estimate that in software engineering people can work as much as 40% faster when under pressure. Such flexibility is very valuable as it enables organizations to accommodate the inevitable, unplanned variations in workload. Both Alpha and Beta adjusted work pressure to keep projects on schedule, although as we will discuss later, Alpha resorted to this approach far more often.

To capture this positive effect of work pressure in **Figure 4** we show the first feedback process in our model, the *Get the Work Done* loop. As the resource ratio grows, pressure increases, leading to a higher rate of *Feature Release,* which, in turn, reduces the stock of *Features Under Development* and therefore alleviates the pressure. All else being equal, this process, a balancing feedback (denoted by the B in the loop's center), works to offset any imbalance between resources and requests, thus enabling development teams to accommodate variations in their workload without constantly adjusting the size of the team or changing the delivery schedule.

Work pressure does also have a dark side: as pressure increases and developers try to work faster in response, they start to make more errors. The increased speed often comes through shortcuts and less attention to detail leading to inadequate requirement development, little documentation of the code, lack of code review, and poor unit testing (see MacCormack, Kemerer, Cusumano et al. 2003 for a quantitative analysis of the effects of different practices on quality and productivity). Moreover, stress induced by sustained pressure can reduce individual productivity and increase absenteeism and turnover, further limiting output. A sustained, high, resource ratio can therefore be counterproductive. In the words of a technical manager in Alpha:

"There is … a lot of pressure put on people to say how can you do this as fast as you can, and people fall back on the ways that they used to do it, so instead of going through all this process stuff [they would say] 'I know what to code, I can just code it up.'"

We capture the negative effects of work pressure by adding a second feedback to our formulation. As work pressure grows, the error rate (broadly defined to include anything that limits net productivity) increases, thereby slowing the net completion rate. As the completion rate slows, the stock of pending features grows (assuming a constant inflow of feature requests) thus feeding back to create even more work pressure. Unlike the "Get the Work Done" loop, this new process, the "Make More Mistakes" loop, is a reinforcing feedback that amplifies any given disturbance. When working in the downward direction— a growing error rate, declining net output, and mounting work pressure—the Rework loop constitutes a vicious cycle.

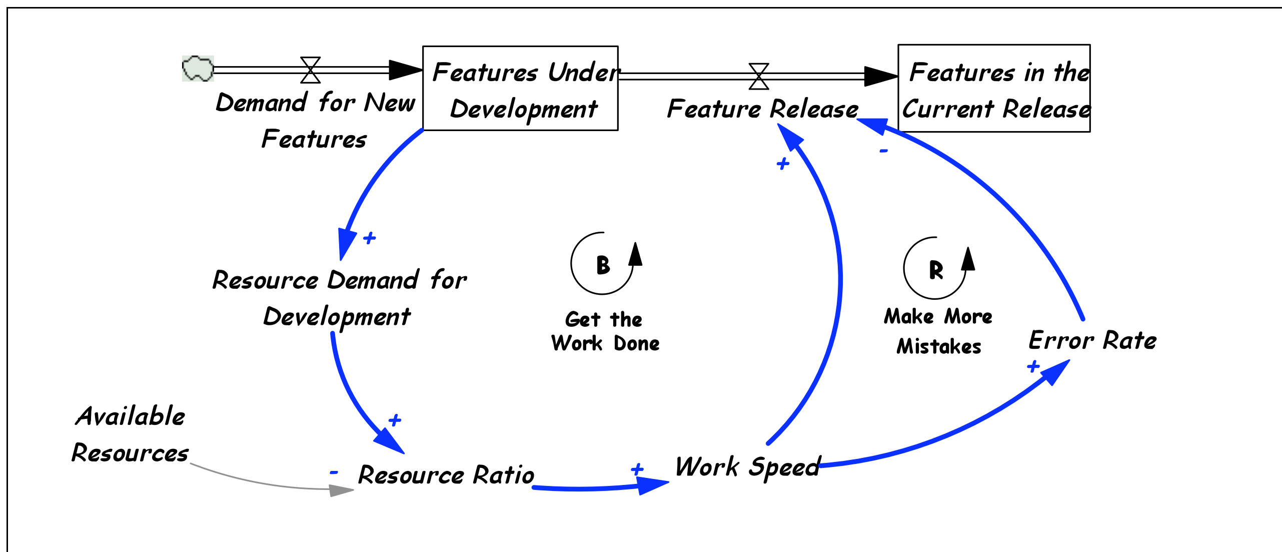


Figure 4- The basic structure of development process and the balancing (Get the Work Done) and reinforcing (Make More Mistakes) loops generated by considering the impact of resource ratio on work speed (gross productivity) and error rate.

Analyzing the dynamics created by the interplay of these two processes requires specifying the net impact of pressure (Resource Ratio) on the rate of Feature Release. Consistent with psychological research on Yerkes-Dodson law (Mandler 1984), and previous research in product

development (Taylor and Ford 2006) and software development (Abdelhamid and Madnick 1991), our discussions with study participants pointed to an inverse U-shape relationship between the resource ratio and net output. At modest levels, work pressure tends to improve net productivity as the positive effect resulting from working faster offsets any increase in the error rate. However, the positive effects of work pressure decline with increased pressure, while the negative effects tend to grow, thus causing the net effect to eventually become negative.

As we mentioned above, while it is derived from a very different set of assumptions, our initial formulation is structurally similar to that described in Rudolph and Repennning (2002), so we only summarize the basic results before moving to our extended formulation. Figure 5-a shows a characterization of the system's dynamics. We begin with the inverted U-shape curve that relates the current resource ratio to the feature-release rate and then characterize the system's dynamics by showing both the system's equilibria and its behavior around those equilibria.

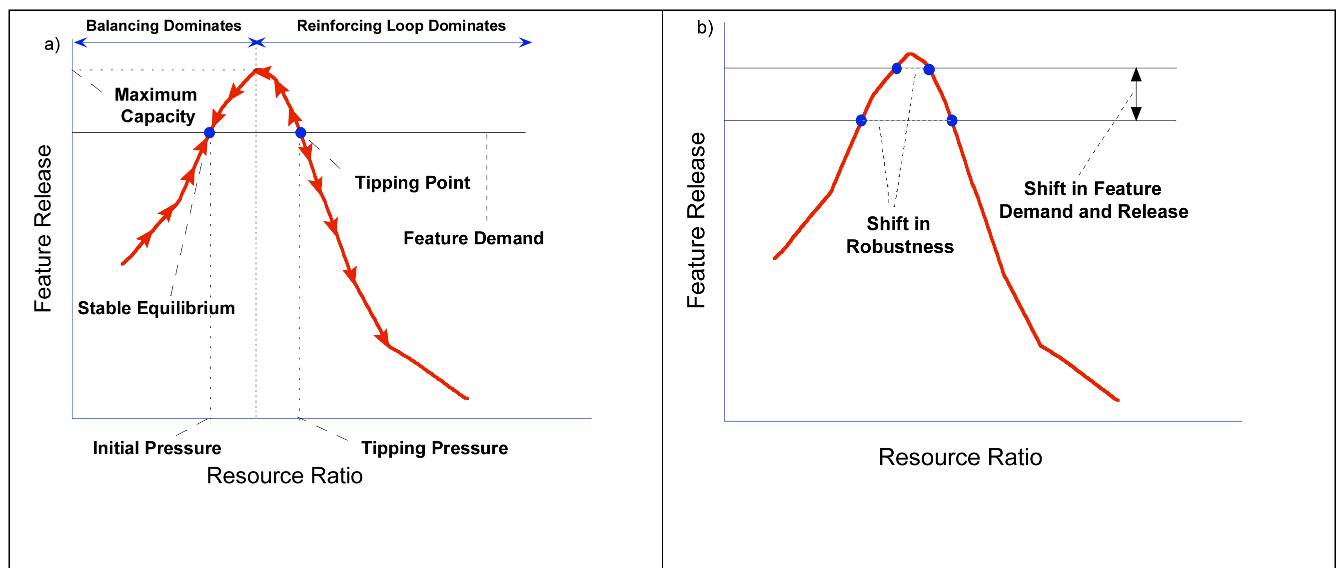


Figure 5- a) Schematic inverse-U shape curve of Feature Release as a function of Resource Ratio. The picture highlights the stable equilibrium, the tipping point, and the maximum capacity. b) Increasing Feature Demand increases feature release (up to maximum capacity); it also reduces robustness, in terms of the shift in Resource Ratio needed to tip the system (the distance between stable equilibrium and tipping point).

The system has two equilibria, each of which occurs at the intersection of the rate of feature demand and the inverted-U. These equilibria represent the points at which the resource ratio induces just enough work effort to meet the current level of feature demand. Once the system reaches either of these points, it will be in steady state (i.e. equilibrium) with the rate of feature release equaling the rate of feature demand. Although both points represent equilibria the dynamics around them are very different. The equilibrium that appears on the upwardly-sloping portion of the inverted-U is *stable*, meaning that small perturbations from it are offset by the system's dynamics. For example, a small increase in the resource ratio (perhaps due to a temporary increase in the stock of features pending) would cause the feature completion rate to increase (manifesting as a temporary excursion off the stable equilibrium and up the inverted U) above the feature demand rate and thus reduce the stock of features pending and bring the system back into balance. Formally, in this region the balancing Getting Work Done loop dominates the system's dynamics and functions exactly as managers desire, constantly working to bring the level of outstanding work and the available resources into balance.

In contrast to the equilibrium in the upwardly sloping portion, the equilibrium that occurs in the downwardly sloping portion of the inverted U is *unstable*, implying that any deviation off that equilibrium, rather than being offset, will be reinforced. Formally, this equilibrium is unstable because it occurs in a region where the system is dominated by reinforcing feedback. As indicated by the arrows on the inverted-U, the system is unlikely to ever settle at this point since

even the smallest perturbation will push the system on to a new trajectory, away from the unstable equilibrium. Due to these dynamics, the unstable equilibrium is often called a *tipping point*, capturing the idea that once an unstable equilibrium is crossed, the system's dynamics can change significantly.

Consistent with this intuition, the tipping point plays a key role in determining the dynamics of the system we study. **Figure 6** shows an experiment in which the Feature Demand rate has been temporarily increased. In the first experiment (**Figure 6**-a) the shock is not sufficient to push the system over the tipping threshold (both experiments start with the system in steady state, operating at the stable equilibrium) and consequently, the system is able to absorb the shock and return to its initial performance level (as measured by the feature release rate). In the second experiment (**Figure 6**-b), in contrast, the increase is large enough to push the system over its tipping threshold. Here, rather than returning to the initial equilibrium, the system gets stuck in a vicious cycle of growing feature demand, mounting work pressure and a declining rate of feature release. As the experiment indicates, when the system crosses its tipping threshold, performance declines significantly and does not recover.

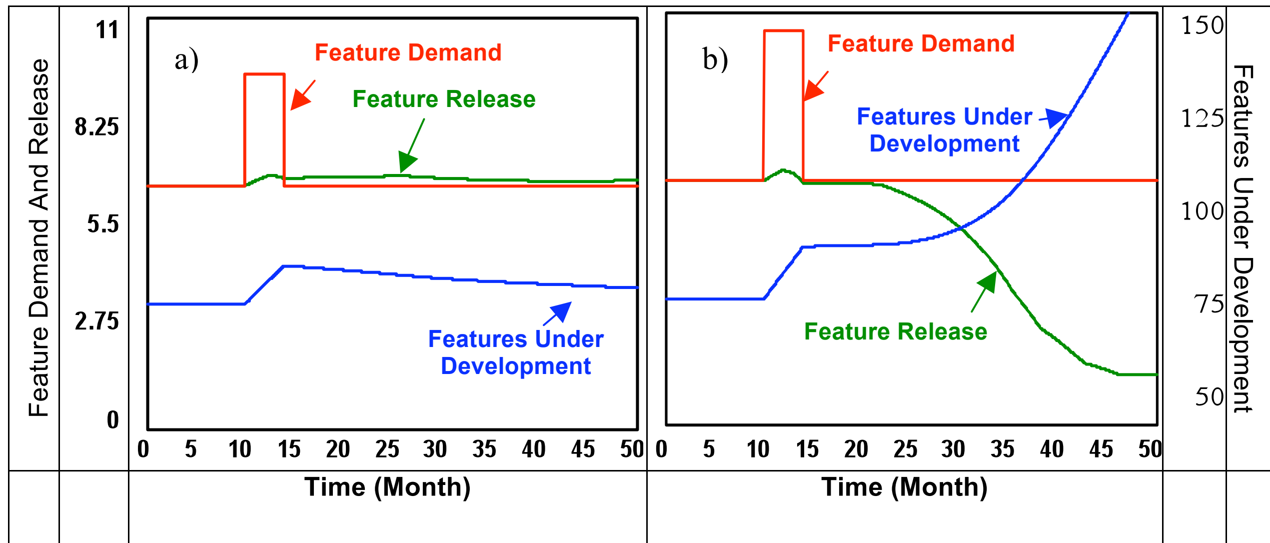


Figure 6- The reaction of simulated PD system to a temporary demand shock under two slightly different shocks. a) The system recovers through the balancing process of "Get the Work Done" b) The system goes over the tipping point, the reinforcing loop "Make More Mistakes" dominates, and system performance deteriorates. Full model formulations are in appendix .

Our characterization highlights three key features of the system's dynamics. First, due to the existence of the tipping threshold, the system's response to temporary shocks is non-linear. Any shock that does not cause the system to cross the tipping threshold is offset, while shocks that do exceed that limit cause a permanent decline in performance. Second, the size of the shock required to push the system over the tipping boundary is determined by the distance between the stable equilibrium and the tipping point. When the distance is large, the system can absorb a substantial shock; when it is small, a modest perturbation can push performance over the edge. Third, the distance between the two equilibria is a function of the steady state rate of feature demand. As shown in Figure 5-b, as the rate of feature demand rises, the two equilbria move up the inverted-U and the distance between them shrinks. If feature demand continues to rise, eventually both equilibria disappear (the horizontal line lies strictly above the inverted-U), and the system displays only one trajectory, a downward spiral into low performance. Thus,

managers in this system face a clear trade-off: As they increase the steady state-level of feature demand, performance continues to improve (as long as they stay within the inverted-U). However, as feature demand rises, the size of the shock required to push the system over its tipping threshold declines.

Our data clearly indicate that Alpha's performance declined over the period of study and the evidence also suggests that they were increasingly trapped in a vicious cycle of accumulating outstanding work, mounting work pressure and declining performance. Moreover, the interviews strongly suggest that the declining performance resulted from changes in the way the work was executed. Several developers noted that, due to schedule pressure, the Alpha team increasingly abandoned the elements of good software practice such as documentation in favor of getting the work done more quickly. And, as the graphs in the previous section indicate, they suffered the consequences. While it started well, Alpha's quality declined significantly and later releases where plagued with bugs requiring additional resources for remediation.

One might have expected that Alpha team would have recognized that they were at the risk of tipping into a vicious cycle and have reduced the feature request accordingly. There is, however, no evidence that such adjustments were ever made. To the contrary, as the tipping model would suggest, their work load only increased over the period of study. To explain why, we turn to the concept of the adaptation trap.

Bug Fixing and Current Engineering. So far we have assumed that errors are discovered within R&D department, and thus affect the system's dynamics only by reducing net

productivity. The real situation in software engineering (and many other development activities) is, however, more complicated. At Sigma some errors escaped the quality assurance process altogether and remained in product that was shipped to customers. When discovered by customers, these errors would be referred back to the R&D department. If the problem interfered with an important customer need, R&D would assign engineers to fix the problem, usually through quick, customized patches at the customer site. These activities, called *current engineering* (CE) can take a significant share of R&D resources: Alpha allocated 30-50 percent of development resources to CE work; Beta, in contrast, kept this number below 10 percent.

Defects, the current engineering they require, and the delays inherent in identifying those resource needs introduce two important wrinkles to our analysis. First, CE requires additional resources. Second, and more importantly, those resource requirements can only be known with a significant delay. Customers must buy, install and use the new release before CE resource needs are revealed. In this section we show how the addition of these two features makes the managers' job far more difficult and the system's descent into firefighting far more likely.

Figure 7 shows the structure of our expanded model. The flow of Feature Release now accumulates in a stock of *Features in the Current Release,* representing the installed base of code released to Sigma customers. A new flow, *Defect Introduction,* now runs in parallel to Feature Release, and captures the stream of defects in new features that are not identified by in-house testing. The defect flow accumulates in the stock of *Defects in the Current Release,* representing the stock of defects in software currently being used by customers.

Defects in the field, as they are eventually discovered, create additional demand for PD resources via current engineering (*Resource Demand for Current Engineering.)* As we show in the diagram, the demand for CE increases the Resource Ratio, potentially creating more pressure and a higher Error Rate. These new links add an additional positive feedback to the model, the reinforcing Current Engineering loop. As the stock of defects grows and customers request additional Current Engineering, developers face even more demands on their time and must work faster to meet their delivery targets. As we discussed in the previous section, however, the increased work speed can result in an even greater error rate, causing the stock of defects in the field to grow further. Note that the strength (or gain) of this loop is potentially high as current engineering is often extremely costly relative to fixing problems before they are released (Boehm 1981). Whereas an error prevented in the design phase can prevent a large number of customer complaints, current engineering activities are often focused on resolving the problem of a single customer. Detailed model equations are available in Appendix A.

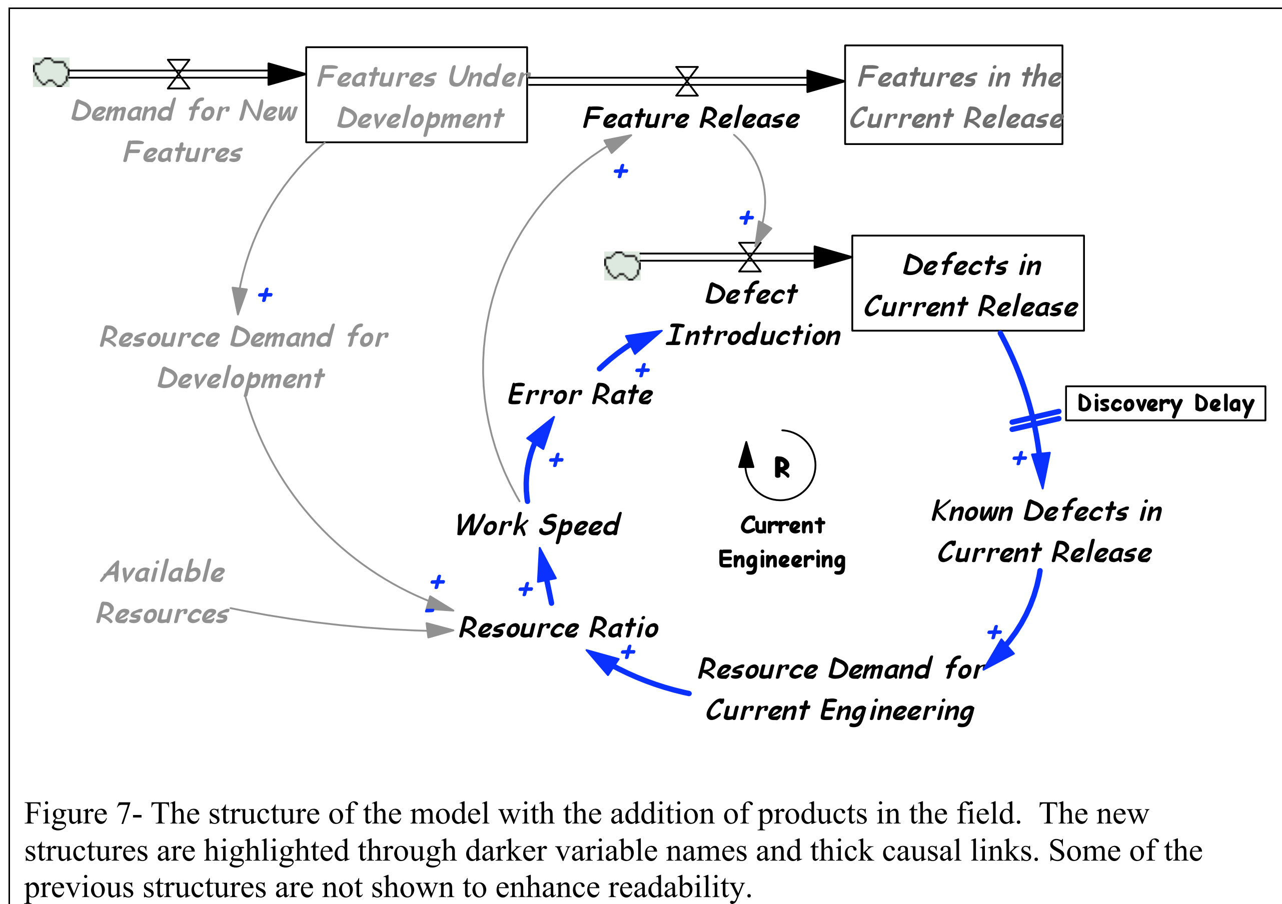


Figure 7- The structure of the model with the addition of products in the field. The new structures are highlighted through darker variable names and thick causal links. Some of the previous structures are not shown to enhance readability.

The addition of current engineering and the delays associated with perceiving its full impact increases the challenge facing those charged with managing resources for two reasons. First, CE requires additional resources and thus reduces the capacity of the organization for developing new features. This shifts the inverse U-shape graph in Figure 5 down. While the existence of the shift is clear, accounting for its magnitude creates a second challenge. Consider the decision to increase the rate of feature demand: Due to the delay in perceiving the increased CE demand, in the short run the system behaves as though no additional errors (from additional features) make it to customer site. Therefore the delay increases the apparent capacity of the system. In fact, as the fraction of errors that go undiscovered grows, so to does the apparent capacity of the system to do additional new work (since resources do not appear to be needed for

rework or additional current engineering) (see Figure 8). In other words, in the short-run the system will behave as though it has more capacity than is actually available.

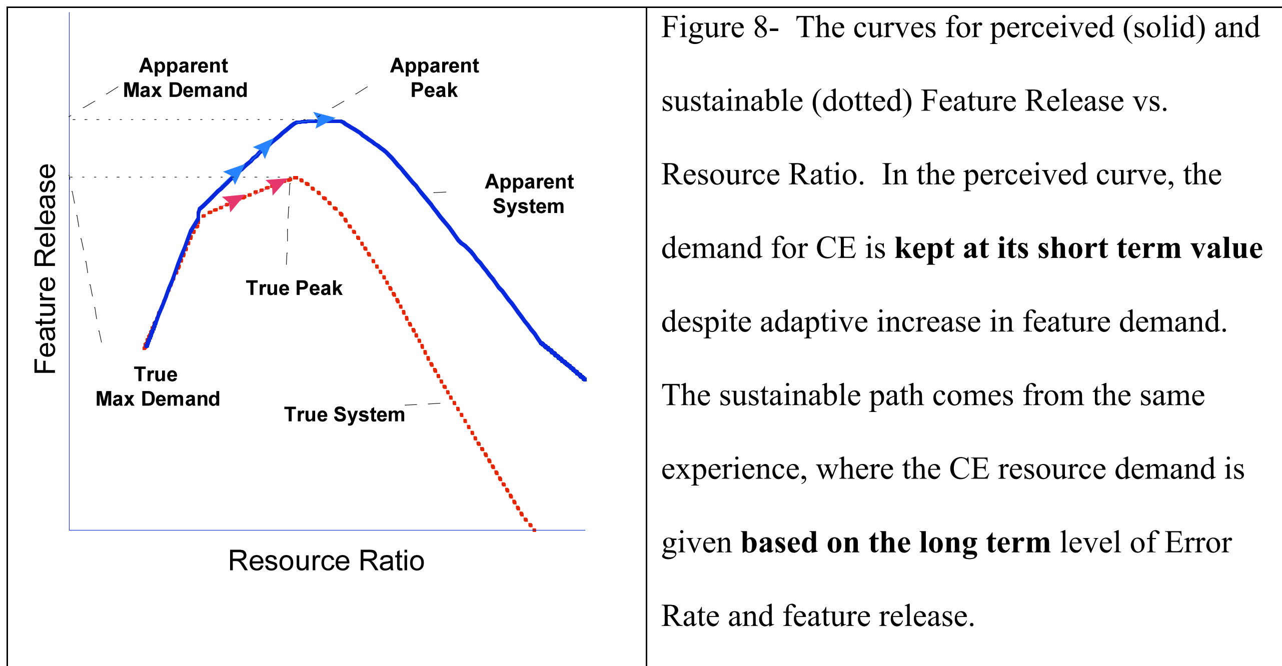


Figure 8- The curves for perceived (solid) and sustainable (dotted) Feature Release vs. Resource Ratio. In the perceived curve, the demand for CE is **kept at its short term value** despite adaptive increase in feature demand. The sustainable path comes from the same experience, where the CE resource demand is given **based on the long term** level of Error Rate and feature release.

This feature of the system—that its short term behavior differs significantly from the long run performance—makes learning difficult. Recall from above that the distance between the stable equilibrium and the tipping threshold is determined by the level of resource utilization. Increasing the rate of feature demand has the effect of pushing both the stable equilibrium and the tipping point "up" the inverted U, reducing the distance between them and making the system more susceptible to firefighting. Thus, the task facing managers of such systems is to set the capacity/resource balance in a way that appropriately trades-off the higher level of performance that comes with a higher level feature demand against the decline in the system's robustness to temporary shocks. Such a decision is not particularly amenable to formal analysis. The organization's true capacity is difficult to estimate as is the workload associated with a given set of features. Managers are thus very likely engage in local search, progressively increasing the

workload and monitoring the performance of the system to determine whether the increases yield a higher rate of feature release. Such local search processes are believed to be one of the principal mechanisms through which organizational routines adapt and create better performance (Nelson and Winter 1982). Tightening the schedule to get the maximum feasible performance was a common strategy in Sigma. A program manager comments:

> *"The thinking that [a higher manager] had, and we think we agree with this, is that if your target is X and everybody is marching for that target, you fill the time for that target. If you are ahead of the game, people are probably going to relax a little bit: because we are ahead of the game, we don't need to work so hard, we can take a longer lunch, we can do whatever; and guess what: Murphy's law says you now won't make that target. So […] you shoot for X minus something, you drive for X minus something so that you build in on Murphy's law to a point."*

The assumption of localized search coupled with a system that displays different short and long run dynamics leads to the notion of the *Adaptation Trap*. As shown in Figure 8, imagine a manager attempting to set the appropriate balance of work and resources by incrementally increasing the rate of feature demand. In the short-run, the system will behave as described by the upper inverted-U (which we label the apparent system). As we show on the graph, such an adaptation path will lead her to choose a level of loading somewhere near the peak of the apparent curve. However, if the true, long run, behavior of the system is described by the lower curve, then the adaptation process will have converged to a rate of feature demand that is beyond the sustainable capacity of the PD organization. When the additional CE work does eventually start to appear, the PD organization will find itself past its tipping threshold and on the slippery slope of firefighting. The tight schedule and high feature demand that initially seemed so effective will cause the rate of feature release to fall below feature demand, leading to continued growth in the stock of features to be developed and, thereby, ensuring that pressure stays high and that the organization remains in the firefighting mode. Thus, in contrast to early experiments

in which an exogenous shock was required to push the system over the edge, the Adaptation Trap notion suggests that such a decline can occur endogenously as the result of the well-intentioned efforts of managers to optimize their system: due to delays in the system, managers do not receive complete information about the results of their experiments, leading them so systematically overload their development system.

The existence of the adaptation traps depends on managers not fully accounting for the delays in the system. Were they to be patient in assessing the full consequence of each incremental increase, they would be unlikely to overshoot. There is however significant experimental and field-based evidence to suggest that this is unlikely. On the experimental side, subjects have been repeatedly shown to both underestimate time delays and fail to properly account for their impact on a system's dynamics (e.g. Sterman 1989; Diehl and Sterman 1995). Similarly, a growing collection of field studies document a similar phenomenon (Oliva and Sterman 2001; Repenning and Sterman 2002). In fact, at Sigma, the calculation of expected resource needs for CE was even less complicated: development teams were expected to dedicate no more than 10% of their workload to CE activities. This was the accepted norm across different PD groups and was implemented resource allocation process for all projects. We observed that its application to planning process was hard to challenge even in the presence of contrary data. For example the same individuals who described the real current engineering load to be over 30% in the case of Alpha, also explained that they planned about 10% of resources for current engineering. Such simple rules are a common and often efficient response in complex and uncertain dynamic systems. Note, however, that the application of such a rule effectively prevents Alpha from ever escaping the firefighting trap. Because they underestimate the resources needed for CE,

sometimes by as much as a factor of five, they take on too much new work, thus insuring that they remain on the wrong side of the tipping threshold.

The strength of the adaptation trap depends on three features of the system, the delay in perceiving the required level of CE, the speed with which managers attempt to optimize resource loading, and the relative productivity of CE versus normal development work. A longer delay in perceiving the demand for CE resources, because of a longer release cycle, will result in a longer time horizon in which the adaptive processes work in the absence of the CE feedback, thus increasing the risk of tipping. Conversely, faster adaptation implies that managers will use less information in determining whether a given change produced a gain in productivity. As managers shrink their evaluation horizon, they account for a smaller portion of the CE effect. Finally, as the productivity of CE falls (relative to normal development work), the trap becomes more likely since each defect that makes it to the field becomes more costly in terms of the foregone development resources. With release cycles (and thus CE feedback delays) of up to one year, resource pressures that pushed for full utilization (thus fast adaptation), and costly customized CE, Alpha faced something of a “perfect storm” in managing its workload.

Although we have so far discussed the adaptation trap as though the project was starting from steady state, often this is not the case. Alpha, for example was a relatively new system, which alone exacerbated the challenge posed by the adaptation trap. In developing the first release of a product there is, by definition, no CE demand. In fact CE demand remains low as long as there are few customers for the product. Thus in its early days, the Alpha team could spend most of its resources on new development activities, leading to a high estimate of the amount of new work it

could accomplish per engineer. As Alpha moved to subsequent releases and sales grew, the demand for current engineering began to increase. In the meantime, Alpha was well received by the market and was installed by several high-profile customers, leading Alpha's management to request additional features so as to maintain Alpha's reputation as an innovative product. Later, as CE requests arrived, the team was careful to respond to them quickly to keep a good relationship with the highly visible customers. The Alpha team thus soon found itself significantly overloaded and was forced to depart from Sigma's normal development practice in an effort to meet the mounting demand for new features and CE. Worse, note how abandoning standard development practice coupled with the delays in discovering CE further constrain the adaptation process: when developers take short cuts, apparent productivity increases because many of the consequent errors will only be discovered when the product reaches the customer site, often several months after that work was completed. Thus the changing work habits of the development team make it more difficult for managers to assess the team's true capacity. As one developer reported:

> *"Nobody really sees what we do. You get rewarded when your name is next to a dollar sign .... Nobody knows what actually the quality is and what actually we do."*

The result of these dynamics is a product that was in part a victim of its own success. Alpha's successful market reception coupled with the delays in assessing the true quality of its code led managers to overestimate the team's capacity for new work. The subsequent workload eventually pushed Alpha over its tipping threshold creating the situation highlighted in the previous section, declining quality and productivity.

Architecture and code base: dynamics of bug-fixing and building on error. In this section we make a final addition to our model to capture the cumulative nature of software development. The analysis so far has been based on the implicit assumption that, from a technical perspective, each release of the software is independent of the previous ones. In other words, although a previous release may steal resources from the current one in the form of current engineering, the execution of a past release has no particular bearing on the work engineers face in this release. In fact, this is not the case. The quality of the past release bears on the execution of the current one in at least two significant ways. First, past errors need to be fixed in a new release, a process know in the industry as bug fixing. Note that bug fixing differs from current engineering as CE focuses on fixing a problem in the software in use at a specific customer site while bug fixing focuses on eliminating the defect entirely in the next release of the software. Second, the quality of documentation and the architecture of the current code base impacts the probability of introducing an error in the current release.

Figure 9 shows these additions in the form of two new feedback processes. First, as the number of Known Defects in Current Release increases, more Defects for Bug Fixing are scheduled for the next release. Not surprisingly, the increased resource demand for bug fixing further worsens the team's workload balance, leading them to work faster and create more errors, thus creating a third reinforcing feedback process. This dynamic was prominent in the later releases of the Alpha project. As in many software projects, the Alpha team distinguished between major and minor releases, with minor releases mainly focused on fixing bugs. As the project continued, the minor releases grew in size, consuming an increasing fraction of the team's time and effort. For example, a recent "minor" release of Alpha included fixing over 1000 bugs, and proved to be as

time- and resource-intensive as any major release. The result, of course, is that the time dedicated to developing the new features necessary to maintain Alpha's market position continued to decline.

Figure 9- The model structure after adding the impact of architecture and code base. New loops are highlighted in thick causal links.

Beyond creating the need for current engineering and bug fixing, errors have an even more insidious effect on productivity and quality. As developers feel increasing pressure to work quickly, they often take shortcuts, both in documentation and in actual coding. These shortcuts can seriously compromise the overall architecture of the software since they often spoil the intended relationships between the different modules of the code. Moreover, the quality of architecture design in the concept design phase of the PD process depends on similar pressures

that impact the development phase. Thus, as the number of defects grow, the ability of developers to predict the effect of a particular change declines. As one informant reported, *"Good architecture control entropy"*, and when the quality of the architecture declines, both because changes have not been carefully analyzed and because they have not been well documented, the chance of introducing an unanticipated interaction increases. We capture this dynamic in the diagram by adding a positive link from the stock of defects in the current release to the *Number of Unanticipated Interactions.* As the number of unanticipated interactions increases, so to does the Error Rate, thus creating the reinforcing Error Creates Error loop.

Introducing bug fixing has an effect similar to that of the previous additions. Bug fixing takes resources away from developing new features, thus shifting the effective, long-term capacity of the PD process further downward. The impact on the evolution of PD capability is also similar: as a product moves to the later releases, the bug-fixing workload increases. Worse, due to the long delays in bug discovery, reporting and assessment, managers will be likely to overestimate the steady state capacity of their development teams (since it takes time to fully account for the effect of low quality). The magnitude of the shift in capacity depends on several factors, including the quality of the code developed (Did we create a lot of bugs?), the strength of testing (Did we fail to catch those bugs initially?), the priority given to fixing bugs, as well as the relative productivity of bug-fixing (Does it take more resources to fix a feature than to develop it correctly in the first place?). In Appendix B we derive the mathematical formula for the sustainable capacity of PD process under different parameter settings.

Besides shifting capacity, the bug-fixing feedback makes the adaptation trap even more seductive. As we discussed above, the delays in assessing the need for bug-fixing activities are long, making adaption more difficult. Moreover, the downward shift in the sustainable capacity of the PD process suggests that the gap between the short-term feature release rate and that which can be sustained in the long-term is wider, further increasing the chance of moving into an unsustainable region.

The *Error Creates Error* feedback is likely to be the last one to become active but it can also be the most detrimental. The loop is likely to be inactive early in a product's life simply because most products start with reasonably well thought-out architectures, as was the case for both Alpha and Beta. Only when the other processes kick in and the initial design is compromised does the architecture begin to degrade. Once that decline does occur, however, it is unlikely that the project will recover. Several subjects reported that poorly documented code with lousy relationships among different modules makes using a high quality development processes increasingly difficult. One of Alpha's developers illustrates this point:

> *"It is scary, the idea that we are working on code but we are afraid to touch too much of it" … "there are so many changes, little features that come in different places, you don't know how they interact with each other [because]… the design is not consistent all the way along, and we didn't have documentation for most of it."*

Thus, not only does the Error Creates Error loop strengthen the effect of the previous feedbacks, further causing developers to abandon high quality practices, but it effectively seals the fate of the project. Reversing the trend requires a significant investment in fixing bugs and refactoring the architecture, and in many cases may prove infeasible. In short, it is hard to save a product line that has fallen so deeply into the firefighting cycle that it faces significant quality problems

resulting from inadequate old architecture and a low quality code base. Therefore the physical constraints of the code and architecture emerging from previous practices affect the routines used in the organization and further erode capability. In fact, in the two years following the end of our data collection, almost all of the work on Alpha has been focused on fixing bugs and very few new features have been added; the prospects for recovery are not good.

The Differing Trajectories of Alpha and Beta

Our analysis suggest that in a well-intentioned effort to maintain its market leadership, the Alpha team failed to accurately assess its true capacity and thus found itself trapped in a vicious cycle of mounting delivery pressure, declining performance, and an increasingly unwieldy and confusing product architecture. Given our earlier claim concerning their similarity, our analysis immediately raises of the question of why did the two projects produce such differing levels of performance? Whereas Alpha suffered a serious decline, Beta continues to be one of the most successful projects at Sigma. Despite their many similarities, Alpha and Beta differed on a few key dimensions, and as we outline in this subsection, those differences, when coupled with the basic structure that we have described so far, caused the two projects to experience very different performance trajectories.

Structural Factors. Our account of the differing performance trajectories of the two projects begins with a key difference between them. While Alpha was a stand-alone product, Beta's delivery was gated by its parent product, a telecommunication switch. Thus, whereas Alpha was directly subjected to delivery pressure from the market, as long as Beta was making more progress than its parent product, the pressure it felt was moderated. Most notably, two times in

Beta's history, the parent product was delayed even though the Beta team was on track to hit the original schedule. Delays in the delivery of the parent product provided the Beta team with time to reflect, improve their processes, and prepare for the next release. Each of these delays pulled Beta farther below its tipping threshold, allowing them to increase the effectiveness of their work. One of the Beta managers reflects on this experience:

> *"How we got here? The answer is in investing in [the] future. And from the [Beta's] perspective we got lucky. If I go back to the first release [of Beta],… they [the parent product] were running into many more schedule problems than we were … this allowed us to go back and actually do a lot of things that we would have preferred to have done at the beginning but we never got the time to do... So we benefited by that additional time to really add a lot of foundational components."*

Note how this advantage, once it tipped towards Beta, could be self-reinforcing; the more the parent product struggled, perhaps due to being in a situation similar to Alpha's, the more time Beta had to focus on high quality work, thereby creating even more space for itself. Alpha, in contrast, benefitted from no such dynamics. It continually faced market pressure and, as its market position slipped due to quality problems, there was additional pressure to add new features to compensate, creating yet another vicious cycle.

Initial Conditions. Alpha and Beta also had different origins. While Beta was developed internally by a team with significant experience, Alpha was born in a start-up organization, subsequently acquired by Sigma, largely staffed by younger, less experienced managers and developers. Their differing origins influenced how the two projects were managed. The Beta team had significant experience managing a product over its full life-cycle and many team members reported having past experience with the costs of skimping on the development disciplines and therefore placed a stronger emphasis on careful adherence to good practice. One of the Beta managers:

> *"Early in my career… when I was in an organization that did wireless… we were under tremendous market pressure to get the product out…and you just do anything you can to get anything out, not one comment was put in a lot of code, not one thing was inspected, not one thing was documented, you know, it was classic, and it never worked, and still to this date it is in the field, and it has problems… and that's were I really cut my teeth and learned my lesson. In [Beta]… we have lots and lots of huge pressures… and we don't skip process, because that is the tone we set… I have had people who skip process and I have had them fired, that it is not a platitude, I insist."*

Alpha, in contrast, originally faced all of the pressures typical of a start up venture. Lacking an ongoing revenue stream and hoping to realize the benefits of first mover advantage, the Alpha team felt significant pressure to get their product to market. And, once that initial release proved successful, the demands for additional features only added to the pressure. These conditions reinforced a culture of "quick and dirty" problem solving, which although possibly helpful in the beginning, proved counter-productive later in the project's life.

Management Policies. The differing initial conditions and external environment combined with the dynamics we have discussed to create two very different approaches to managing software development. Discussions with Beta's managers suggested that many of them had an intuitive sense of the dynamics we discuss and thus actively worked to avoid them. Notably, Beta displayed several organizational practices targeted at maintaining quality that were not present in Alpha. For example, root cause analyses of problems reported from the field were often conducted in Beta, and the results could impact personnel review and incentives. The developers therefore knew that if they cut corners and things went wrong, they would be held responsible when bugs surfaced in the field. Similarly, the Beta organization instituted a "No-Late-Feature" rule, which was enforced by the team's leadership. Whereas Alpha was frequently subjected to changes in requirements far into the design phase, Beta's managers were typically able to resist

the pressure to add features once design started. The rule helped avoid spikes in workload not captured in the original plan and thus played a central role in keeping Beta below its tipping threshold. Finally, in contrast to Alpha, Beta had a separate, dedicated team of developers focused on current engineering. By not allowing the increase in CE demand to impact the development resources available to the next release this policy effectively breaks the current engineering loop discussed above.

At a first glance, it might appear that, in contrast to their colleagues in the Beta project, Alpha's managers were simply naïve and unskilled. A closer examination suggests that the situation is more complicated. Alpha's managers reported being able to see the benefits of Beta's practices (which reflected industry best practice), but they found them infeasible for their own organization. For example, root-cause analysis is very time consuming and only possible if the project faces a relatively small stream of bugs coming from the field. While Beta was able to do a full root cause analysis for many of its bugs, Alpha's bug stream was more than an order of magnitude larger, thus making comprehensive root-cause-analysis infeasible.

Attempting to separate the resources dedicated to CE creates similar challenges. CE demand is uncertain and hard to plan. When the CE demand is low, it is possible to dedicate a small team to that work and accept the inevitable idle time that occurs when no new bugs are reported. When CE work grows to over 30% of the total resource load (as it did in Alpha), it becomes prohibitively expensive. The organization simply cannot bear the costs of keeping a significant fraction of its people idle at the times that demand is low. Both examples suggest that team's approach to managing the project tended to co-evolve with the state of the project itself. Beta's

stability tended to be self-reinforcing, creating more space for the investments necessary to maintain best practice. Alpha, in contrast, faced increasing pressure for short-term delivery, causing it to progressively abandon high performance practice.

The "No-Late-Feature" rule provides an example of how this co-evolution spreads beyond the internal workings of the team to the surrounding organization. Several informants reported that one of the main reasons that the Beta team could resist the pressure of senior management and customers to add late features was the reputation and bargaining power gained from their history of high quality and on-time delivery. In absence of such credibility and bargaining power, Alpha's managers found it far more difficult to resist late feature requests, which they generally agreed were detrimental to the project's performance.

Thus, although both teams were familiar with the disciplines that constituted effective software development, as the two projects progressed, the teams' abilities to put those disciplines into practice diverged. Whereas the Beta project continued to follow best practice, Alpha found itself increasingly trapped in a vicious cycle of mounting quality problems, increasing pressure for additional productivity and declining adherence to best practice.

DISCUSSION

Comparing the experience of Alpha and Beta thus raises a key question for students of strategy and organization: what is the relevant capability that Beta managed to maintain but Alpha did not? Consistent with previous work (e.g. Klein and Sorra 1996; Pfeffer, Sutton et al. 2000; Repenning and Sterman 2002) our data suggest that knowledge of the specific practices did not,

in and of itself, constitute a capability. Alpha's managers were familiar with the activities that constituted good software practice and frequently lamented their inability to use them. Moreover, members of Sigma's central development organization had in some cases literally written the book on good software development practice; *knowing* the right thing to do, while necessary, was not sufficient to produce high performance in every project.

Comparing Alpha and Beta suggests that the ability to translate knowledge into effective practice both turned on the state of the project in question and tended to be self-reinforcing. The relatively high quality of the Beta product afforded the development team both the space and the authority to follow best practice and thus maintain the quality of their product delivery. In contrast, Alpha's troubled performance eroded the team's power to resist the addition of features and increasingly pushed the team to abandon the disciplines associated with good practice. Such reinforcing processes appear to be at the heart of differing capability trajectories.

Identifying the positive feedbacks at the heart of capability development and erosion is, however, only part of the answer. What policies or structures dictated whether these processes worked in an upward, virtuous direction, as in Beta, or a downward, vicious direction, as in Alpha? Our search for the answer to this question returned us to a notion at the very foundation of organization theory: the ability to balance resources and demands. At the core of our account of the two differing trajectories lies one specific decision point, the amount of work undertaken given the available resources. Analyzing our model suggests that when teams try to take on too much work, they run a significant risk of tipping into a pathological regime in which it is increasingly difficult to follow good development practice. While central to early accounts of

organizational behavior (Pfeffer and Salancik 1978), the notion that organizations should balance resources and demands has received little attention in the recent literature. Despite the lack of scholarly attention, our analysis suggests that under conditions to be outlined momentarily, balancing resources and demands is central to allowing an operational capability to flourish. Our data suggest that when organizations take on more work than they have the resources to accomplish, participants change their behaviors in ways that cause capability to erode. And, although it has not been previously highlighted as a capability, the criticality of resource balancing in insuring the continued use of the best practices has been highlighted in several previous studies (e.g. Repenning 2001; Taylor and Ford 2006).

The first contribution of our study is to highlight the ability to balance resource and demands as a *foundational capability*. The skill and ability to write quality code is also certainly a capability, but our data suggest that it is less relevant in explaining the differences in performance than the contextual and managerial factors that explain its use or lack thereof. Distinguishing between so-called zero-level capabilities—the ability to develop code—and foundational capabilities—the managerial skill and foresight to balance resources appropriately —draws attention away from specific practices and technologies, which vary widely across industries, to basic managerial and administrative functions. Redirecting attention in this manner may help strategy scholars hone in upon a more general set of principles from which to understand the sources of competitive advantage.

Highlighting the criticality of resource balancing as a foundation on which capability is developed does appear at odds with some of the current literature on high performance

organizations, particularly software development. Recently, some authors have argued for the utility of methods such as agile development in which careful planning and resource balancing are scrapped in favor of rapid iteration. In such methods, development teams simply work through as many build-test cycles as is possible prior to the release of the project and planning plays a relatively minor role. Anecdotal evidence suggests organizations following these methods can also produce above average performance (Martin 2002; Larman 2003).

Within the resolution for these competing views lies the second major contribution of our study. Our efforts have identified a specific, foundational capability that may eventually be shown to be a general source of competitive advantage. That said, we do not claim that resource balancing will be strategically important in all contexts. To the contrary, our formal analysis yields a clear set of conditions under which the pathological dynamics we study are likely to appear. As such, our analysis yields three sharp hypotheses concerning when resource balancing will constitute a strategically important capability. In other words, when the structural conditions necessary for the reinforcing dynamics we study are not present, our analysis suggests that resource balancing will be relatively less important and therefore would not lead to superior performance.

The first of these structural conditions concerns the role of work quality in creating additional demands on the organization. The tipping dynamics we study turn heavily on the fact that low quality work fed back to create additional work and put additional pressure on the development team. When this assumption is not satisfied, the positive loops at the center of our analysis do not exist. Thus, the first hypothesis emerging from our analysis is that the value of resource balancing is positively associated with the resource requirements resulting from poor work.

In the cases of Alpha and Beta the link between poor work and subsequent resource requirements was strong; each defect could create a significant demand for both current engineering and bug fixing. Not all development activities, however, satisfy this condition. For example, in many industrial settings, development teams simply have to live with the defects in the product they create and have few opportunities to fix them. Toys and fashion goods are examples. Although defects in the design may cost the company money in terms of sales and/or warranty expense, the designers and engineers in this context are unlikely to be distracted from their work on the next generation since the designs are almost entirely new from one year to the next. In contrast, health care, where resources must be allocated between preventive care and acute care, is likely to exhibit a dynamic similar to that at Sigma: lack of attention to preventive care, with a long delay, leads to major costs in acute care, further distracting resources from preventive care.

The second condition concerns the fungibility of resources between doing work and fixing the problems arising from past efforts. The reinforcing feedbacks we study were particularly strong because it was relatively easy for the development teams to shift between new work and fixing bugs. When resources are less fungible, those dynamics will be muted. Thus our second hypothesis suggests that resource balancing will yield larger gains in performance when resources are fungible. Much like the first hypothesis, software development is a strong case in point. Project resources are often perfectly fungible, with development teams working on everything from developing new features to fixing issues at specific customer sites. Heavy manufacturing, such as car production, represents an intermediate point on the fungibility continuum. When problems appear in the field, engineers need to develop a fix and insure that

the defect is eliminated in future production. In contrast to Alpha and Beta, however, engineers in this context rarely go fix individual automobiles. Finally, consider the case of developing a feature film or a high fashion product. In these cases, while defects originating from poor development will be expensive, such problems are not likely to draw on the resources that created them. Filmmakers and designers do not redo their products when they are poorly received by the market; that burden falls to their respective marketing departments.

The final condition concerns the delay between performing work and perceiving its true quality. In our analysis, this delay proves central in explaining the drift towards low performance. Because managers cannot easily anticipate future resource requirements resulting from defective work, they are prone to taking on too much work, thereby getting stuck in the adaptation trap. Thus the final hypothesis emerging from our analysis is that the value of resource balancing will rise as the time required to perceive resource requirements increase.

Here Sigma probably represents an intermediate case. Development managers at Sigma were able to see the quality of their teams' work over the course of six to twelve months. The delays are much longer in other industries. In the auto industry, for example, in can take several years to develop a new vehicle and then at least one more year before the true quality of that vehicle has been revealed (and some problems can take far longer). On the other side of the continuum, website developers (frequent proponents of agile development methods) can often get feedback on their work in a matter of days and easily make adjustments. Our analysis suggests that resource balancing is a key capability potentially explaining firm heterogeneity in the auto industry but of little explanatory value in developing websites.

Our analysis thus suggests a contingent view of the value of resource balancing and in doing so provides a potential resolution to an ongoing debate in the management literature. For firms whose products tend to have long lifetimes and designs that build upon previous releases, resource balancing will be a key source of competitive advantage. In contrast, for firms whose products are less cumulative, subjected to faster learning, and whose defects do not generate additional work, resource planning will be less relevant as a core capability. In fact, for these products, resource balancing may actually impede performance as it require formal processes that reduce flexibility and the ability to adapt to changing market conditions.

Our work has a broader implication for managers and theorists. While early students of organizational behavior recognized that resource balancing was a key function of the executive, it appears that the scholarly and practitioner communities have subsequently lost sight of this function. Three forces appear to be responsible. First, as argued by Barley and Kunda (Barley and Kunda 2001), organizational scholarship has grown more distant from the nuts and bolts details of getting work done and few activities appear more routine than matching work to resources. Thus, scholars have chosen to focus on more macro level phenomena. A second and related trend is the rise of structural functionalism and economic-based explanations (Perrow 1986; Pfeffer 1993). The dynamics we identify are fundamentally pathological and would be extremely difficult (although probably not impossible) to explain with a rational or functional approach. Thus, it appears, that many have assumed that resources must be roughly in balance and not worthy of further attention. There is growing evidence however that many organizations take on far more work than they are capable of doing.

Third, this blind spot does not appear to be unique to scholars. To the contrary, management practitioners (at least those is western companies) appear to grow ever more obsessed with individual accountability and many view overload, often phrased as a "stretch objective", as a key tool in insuring individual performance. As in Sigma, many managers now believe that the worst thing that can happen if they put too much work into the system is that their people will be fully utilized.

Ironically, this view has been thoroughly discredited in the world of discrete manufacturing. Perhaps the most significant lesson of the so-called lean revolution is that attempting to run a factory above its capacity is a sure strategy for maintaining low performance. But, while this view has been widely accepted on the factory floor, with a few exceptions, it has yet to permeate other types of work. And, there is a credible argument that the costs of this misconception are significant. A crushing workload and time pressure have been implicated in essentially every major industrial accident in the last twenty years including the two space shuttle disasters and the explosion at the BP's Texas City refinery (MacKenzie, Holmstrom et al. 2007). Similarly, the growing collection of studies focused on medical error reach a similar conclusion, too many patients being served by too few doctors and nurses is a recipe for disaster. Whether these arguments will survive empirical test is an open question. But, the existing data suggests they provide both a fruitful area for new work and, more importantly, an opportunity for research to make a significant contribution to practice.

References:

Abdelhamid, T. and S. E. Madnick (1991). Software project dynamics: An integrated approach. Englewood Cliffs, NJ, Prentice-Hall.

Argote, L. and D. Epple (1990). "Learning-Curves in Manufacturing." Science **247**(4945): 920-924.

Barley, S. and G. Kunda (2001). "Bringing Work Back In." Organization Science **12**: 76-95.

Barney, J. (1991). "Firm Resources and Sustained Competitive Advantage." Journal of Management **17**(1): 99-120.

Boehm, B. W. (1981). Software engineering economics. Englewood Cliffs, N.J., Prentice-Hall.

Diehl, E. and J. Sterman (1995). "Effects of Feedback Complexity on Dynamic Decision Making." Organizational Behavior and Human Decision Processes **62**(2): 198-215.

Dierickx, I. and K. Cool (1989). "Asset Stock Accumulation and Sustainability of Competitive Advantage." Management Science **35**(12): 1504-1511.

Eisenhardt, K. M. (1989). "Building theories from case study research." Academy of Management Review **14**(4): 532-550.

Eisenhardt, K. M. and J. A. Martin (2000). "Dynamic capabilities: What are they?" Strategic Management Journal **21**(10-11): 1105-1121.

Ethiraj, S. K., P. Kale, M. S. Krishnan and J. V. Singh (2005). "Where do capabilities come from and how do they matter? A study in the software services industry." Strategic Management Journal **26**(1): 25-45.

Fisher, S. (1986). Stress and Strategy. London, Lawrence Erlbaum.

Forrester, J. W. (1961). Industrial Dynamics. Cambridge, The M.I.T. Press.

Gavetti, G. (2005). "Cognition and hierarchy: Rethinking the microfoundations of capabilities' development." Organization Science **16**(6): 599-617.

Glaser, B. G. and A. L. Strauss (1967). The discovery of grounded theory; strategies for qualitative research. Chicago,, Aldine Pub. Co.

Helfat, C. E. (2000). "Guest editor's introduction to the special issue: The evolution of firm capabilities." Strategic Management Journal **21**(10-11): 955-959.

Helfat, C. E. and M. A. Peteraf (2003). "The dynamic resource-based view: Capability lifecycles." Strategic Management Journal **24**(10): 997-1010.

Henderson, R. and I. Cockburn (1994). "Measuring Competence - Exploring Firm Effects in Pharmaceutical Research." Strategic Management Journal **15**: 63-84.

Henderson, R. M. and K. B. Clark (1990). "Architectural Innovation - the Reconfiguration of Existing Product Technologies and the Failure of Established Firms." Administrative Science Quarterly **35**(1): 9-30.

Jones, C. (2000). Software assessments, benchmarks, and best practices. Boston, MA, Addison-Wesley.

Kaplan, S. and R. Henderson (2005). "Inertia and incentives: Bridging organizational economics and organizational theory." Organization Science **16**(5): 509-521.

Keil, T. (2004). "Building external corporate venturing capability." Journal of Management Studies **41**(5): 799-825.

Klein, K. J. and J. S. Sorra (1996). "The challenge of innovation implementation." Academy of Management Review **21**(4): 1055-1080.

Larman, C. (2003). Agile and Iterative Development: A Manager's Guide, Addison-Wesley Professional.

Lavie, D. (2006). "Capability reconfiguration: An analysis of incumbent responses to technological change." Academy of Management Review **31**(1): 153-174.
Leonard-Barton, D. (1992). "Core Capabilities and Core Rigidities - a Paradox in Managing New Product Development." Strategic Management Journal **13**: 111-125.
Levinthal, D. A. (1997). "Adaptation on rugged landscapes." Management Science **43**(7): 934-950.
Levitt, B. and J. G. March (1988). "Organizational Learning." Annual Review of Sociology **14**: 319-340.
MacCormack, A., C. F. Kemerer, M. Cusumano and B. Crandall (2003). "Trade-offs between productivity and quality in selecting software development practices." Ieee Software **20**(5): 78-+.
MacKenzie, C., D. Holmstrom and M. Kaszniak (2007). "Human Factors Analysis of the BP Texas City Refinery Explosion." Human Factors and Ergonomics Society Annual Meeting Proceedings, Safety **5**: 1444-1448.
Mandler, G. (1984). Mind and body : psychology of emotion and stress. New York, W.W. Norton.
Martin, R. C. (2002). Agile Software Development, Principles, Patterns, and Practices, Prentice Hall.
Mauri, A. J. and M. P. Michaels (1998). "Firm and industry effects within strategic management: An empirical examination." Strategic Management Journal **19**(3): 211-219.
Nelson, R. R. and S. G. Winter (1982). An evolutionary theory of economic change. Cambridge, Mass., Belknap Press of Harvard University Press.
Oliva, R. and J. D. Sterman (2001). "Cutting corners and working overtime: Quality erosion in the service industry." Management Science **47**(7): 894-914.
Perrow, C. (1986). Complex organizations : a critical essay. New York, Random House.
Peteraf, M. A. (1993). "The Cornerstones of Competitive Advantage - a Resource-Based View." Strategic Management Journal **14**(3): 179-191.
Pfeffer, J. (1993). "Barriers to the advance of organizational science: Paradigm development as a dependent variable." Academy of Management Review **18**: 599-620.
Pfeffer, J. and G. R. Salancik (1978). The external control of organizations : a resource dependence perspective. New York, Harper & Row.
Pfeffer, J., R. I. Sutton and NetLibrary Inc. (2000). The knowing-doing gap
how smart companies turn knowledge into action. Boston, Mass., Harvard Business School Press**:** xv, 314 p.
Porter, M. E. (1998). Competitive strategy : techniques for analyzing industries and competitors : with a new introduction. New York, Free Press.
Priem, R. L. and J. E. Butler (2001). "Is the resource-based "view" a useful perspective for strategic management research?" Academy of Management Review **26**(1): 22-40.
Raff, D. M. G. (2000). "Superstores end the evolution of firm capabilities in American bookselling." Strategic Management Journal **21**(10-11): 1043-1059.
Rahmandad, H. (2005). Three essay on modeling dynamic organizational processes. Sloan School of Management. Cambridge, Massachusetts Institute of Technology. **Ph.D.**
Repenning, N. P. (2001). "Understanding fire fighting in new product development." The Journal of Product Innovation Management **18**: 285-300.

Repenning, N. P. and J. D. Sterman (2002). "Capability Traps and Self-Confirming Attribution Errors in the Dynamics of Process Improvement." Administrative Science Quarterly **47**: 265-295.

Rivkin, J. W. (2000). "Imitation of complex strategies." Management Science **46**(6): 824-844.

Rivkin, J. W. (2001). "Reproducing knowledge: Replication without imitation at moderate complexity." Organization Science **12**(3): 274-293.

Roberts, P. W. and G. R. Dowling (2002). "Corporate reputation and sustained superior financial performance." Strategic Management Journal **23**(12): 1077-1093.

Rosenbloom, R. S. (2000). "Leadership, capabilities, and technological change: The transformation of NCR in the electronic era." Strategic Management Journal **21**(10-11): 1083-1103.

Rudolph, J. W. and N. P. Repenning (2002). "Disaster Dynamics: Undrstanding the Role of Quantity in Organizational Collapse." Administrative Science Quarterly **47**: 1-30.

Rumelt, R. P. (1991). "How Much Does Industry Matter." Strategic Management Journal **12**(3): 167-185.

Schreyogg, G. and M. Kliesch-Eberl (2007). "How dynamic can organizational capabilities be? Towards a dual-process model of capability dynamization." Strategic Management Journal **28**(9): 913-933.

Siggelkow, N. (2002). "Evolution toward fit." Administrative Science Quarterly **47**(1): 125-159.

Spanos, Y. E. and S. Lioukas (2001). "An examination into the causal logic of rent generation: Contrasting Porter's competitive strategy framework and the resource-based perspective." Strategic Management Journal **22**(10): 907-934.

Sterman, J. (2000). Business Dynamics: systems thinking and modeling for a complex world. Irwin, McGraw-Hill.

Sterman, J. D. (1989). "Modeling Managerial Behavior: Misperceptions of Feedback in a Dynamic Decision Making Experiment." Management Science **35**(3): 321-339.

Sterman, J. D. (1994). "Learning in and about complex systems." System Dynamics Review **10**(2-3): 91-330.

Sterman, J. K., N. P. Repenning and F. Kofman (1997). "Unanticipated side effects of successful quality programs: Exploring a paradox of organizational improvement." Management Science **43**: 503-521.

Taylor, T. and D. N. Ford (2006). "Tipping point failure and robustness in single development projects." System Dynamics Review **22**(1): 51-71.

Teece, D. J., G. Pisano and A. Shuen (1997). "Dynamic capabilities and strategic management." Strategic Management Journal **18**(7): 509-533.

Tripsas, M. and G. Gavetti (2000). "Capabilities, cognition, and inertia: evidence from digital imaging." Strategic Management Journal **21**: 1147-1161.

Wernerfelt, B. (1984). "A Resource-Based View of the Firm." Strategic Management Journal **5**(2): 171-180.

Williamson, O. E. (1999). "Strategy research: Governance and competence perspectives." Strategic Management Journal **20**(12): 1087-1108.

Winter, S. G. (2000). "The satisficing principle in capability learning." Strategic Management Journal **21**(10-11): 981-996.

Winter, S. G. (2003). "Understanding dynamic capabilities." Strategic Management Journal **24**(10): 991-995.

Zollo, M. and S. G. Winter (2002). "Deliberate learning and the evolution of dynamic capabilities." Organization Science **13**(3): 339-351.

Appendix A- Detailed model formulations

The following table lists the equations for the full model. To avoid complexity, all switches and parameters used to break feedback loops are removed from the equations.

Allocated Bug Fix Resources = Allocated Resources[BugFix]   Units: Person
Allocated Resources[Function] = Min(Desired Resources[Function]/SUM(Desired Resources[Function!])*Total Resources, Desired Resources[Function])   Units: Person Desired Resources[Development] = Des Dev Resources Desired Resources[CurrentEng] = Desired Resources for Current Engineering Desired Resources[BugFix] = Desired Bug Fix Resources Development Resources Allocated = Allocated Resources[Development] CE Resources Allocated = Allocated Resources[CurrentEng] Allocated Bug Fix Resources = Allocated Resources[BugFix]
Average size of a new release = 40   Units: Feature
Bug Fixing = Allocated Bug Fix Resources*Productivity of Bug Fixes   Units: Feature/Month
Des Dev Resources = Features Under Development / Desired Time to Develop / Productivity / (1-Normal Error Rate*Frac Error Caught in Test)   Units: Person
Desired Bug Fix Resources = Errors in Base Code / Time to Fix Bugs / Productivity of Bug Fixes   Units: Person
Desired Resources for Current Engineering = Sales*(Errors in New Code Developed+Effective Errors in Code Base)*Fraction Errors Showing Up / Productivity of CE Units: Person

Desired Time to Develop = 10 Units: Month

Dev Work Pressure = if then else (Resource Pressure Deficit < 0.99, Resource Pressure Deficit, ZIDZ (Des Dev Resources , Development Resources Allocated)) Units: Dmnl

Eff Architecture and Code Base on Error Rate = (Modularity Coefficient*"Tl Eff Arc & Base on Error"(Fraction Old Features Defective)+(1-Modularity Coefficient)*Max (1, Errors in Base Code)) Units: Dmnl

Eff Pressure on Error Rate = Tl Eff Pressure on Error Rate (Dev Work Pressure) Units: Dmnl

Effect of Pressure on Productivity = Tl Eff Pressure on Productivity (Dev Work Pressure) Units: Dmnl

Effective Errors in Code Base = Min (Effective Size of Old Release Portion , Old Features Developed)*Fraction Old Features Defective Units: Feature

Effective Size of Old Release Portion = 0 Units: Feature

Error Becoming Old = Features Becoming Old*Fraction New Features Defective Units: Feature/Month

Error Frac in Released Feature = Error Fraction*(1-Frac Error Caught in Test) / (1-Error Fraction*Frac Error Caught in Test) Units: Dmnl

Error Fraction = Min (1, Normal Error Rate*Eff Architecture and Code Base on Error Rate*Eff Pressure on Error Rate) Units: Dmnl

Error Generation = Feature Release*Error Frac in Released Feature Units: Feature/Month

Errors in Base Code = INTEG(Error Becoming Old-Bug Fixing , Errors in New Code Developed*Time to Fix Bugs / Desired Time to Develop) Units: Feature

Errors in New Code Developed = INTEG(Error Generation-Error Becoming Old , Normal

Error Rate*(1-Frac Error Caught in Test)*New Features Developed / (1-Normal Error Rate*Frac Error Caught in Test)) Units: Feature
External Shock = 0 Units: Feature
Feature Addition = Fixed Feature Addition+PULSE (Pulse Time , Pulse Length)*External Shock / Pulse Length Units: Feature/Month
Feature Release = Rate of Development*Fraction of Work Accepted Units: Feature/Month
Features Becoming Old = New Features Developed / Time Between Releases Units: Feature/Month
Features Under Development = INTEG(Feature Addition-Feature Release , Fixed Feature Addition*Desired Time to Develop) Units: Feature
FINAL TIME = 100 Units: Month
Fixed Feature Addition = 4 Units: Feature/Month
Frac Error Caught in Test = 0.9 Units: Dmnl
Fraction Errors Showing Up = 0.1 Units: Dmnl
Fraction New Features Defective = ZIDZ (Errors in New Code Developed , New Features Developed) Units: Dmnl
Fraction of Work Accepted = 1-Error Fraction*Frac Error Caught in Test Units: Dmnl
Fraction Old Features Defective = ZIDZ (Errors in Base Code , Old Features Developed) Units: Dmnl
Modularity Coefficient = 0.5 Units: Dmnl
New Features Developed = INTEG(Feature Release-Features Becoming Old , Average size of a new release) Units: Feature
Normal Error Rate = 0.4 Units: Dmnl

Old Features Developed = INTEG(Features Becoming Old , Initial Old Features) Units: Feature
Productivity = 0.1 Units: Feature/(Person*Month)
Productivity of Bug Fixes = Productivity*Relative Producitivty of Bug Fixing Units: Feature/(Person*Month)
Productivity of CE = Productivity*Relative Productivity of CE Units: Feature/(Person*Month)
Pulse Length = 1 Units: Month
Pulse Time = 10 Units: Month
Rate of Development = Development Resources Allocated*Productivity*Effect of Pressure on Productivity Units: Feature/Month
Relative Producitivty of Bug Fixing = 0.5 Units: Dmnl
Relative Productivity of CE = 1 Units: Dmnl
Resource Pressure Deficit = SUM (Allocated Resources[Function!]) / Total Resources Units: Dmnl
Sales = 10 Units: 1/Month
Time Between Releases = Average size of a new release / Feature Release Units: Month
TIME STEP = 0.125 Units: Month
Time to Fix Bugs = 9 Units: Month [0,20]
"Tl Eff Arc & Base on Error" ([(0,0)-(1,2)],(0,1),(0.13,1.13),(0.28,1.34),(0.42,1.68),(0.54,1.84),(0.72,1.94),(1,2)) Units: Dmnl
Tl Eff Pressure on Error Rate ([(0.5,0)-(2,2)],(0.5,0.8),(1,1),(1.15,1.11) ,(1.29,1.33),(1.40,1.54),(1.53,1.74),(1.68,1.89),(1.99,2)) Units: Dmnl

Tl Eff Pressure on Productivity ([(0,0)-(2,1.5)],(0,0.6),(0.26,0.61),(0.45,0.64) ,(0.64,0.73),(0.83,0.84),(1,1),(1.22,1.2),(1.43,1.28),(1.77,1.32),(2,1.33)) Units: Dmnl
Total Resources = 100 Units: Person

Appendix B- The maximum PD capacity

With a few assumptions, we can draw the analytical expressions for long-term capacity of the product development organization as modeled in the essay. These assumptions include:

- That products remain in the field for a fixed time
- Product modularity is not very high, therefore we need to keep the absolute level of problems in the code base and architecture low.

The logic for deriving the capacity is simple. We can write the total resources needed for development, CE, and bug-fixing in terms of Feature Release and Resource Ratio. Since in this model, allocation is proportional to request, the ratio of resources requested to those allocated are the same across the three functions and equal resource ratio. Therefore we can find the resources allocated to each activity in terms of resource ratio and Feature Release. Consequently, knowing that when resource ratio is above 1, the total resources allocated equal the total resources available, we can find Feature Release in terms of resource ratio, which is the equation we need to find the optimum capacity.

Formally, we can write resources allocated to each activity:

$$R_D = \frac{FR}{f_P(RG).P_N.(1 - e_N.f_e(RG).f_r)} \quad (3)$$

$$R_{CE} = \frac{FR.e_N.f_e(RG).(1 - f_r).k}{P_N.(1 - e_N.f_e(RG).f_r).RP_{CE}} * \frac{1}{RG} \quad (4)$$

$$R_{BF} = \frac{FR.e_N.f_e(RG).(1 - f_f)}{P_N.(1 - e_N.f_e(RG).f_f).RP_{BF}} * \frac{1}{RG} \qquad (5)$$

And note the conservation of resources:

$$R = R_D + R_{CE} + R_{BF} \qquad (6)$$

Plugging the equations 3-5 into equation 6, and solving for FR, we find the following relationship that describes Feature Release as a function of resource ratio:

$$FR = \frac{R.P_N.(1 - e_N.f_e(RG).f_f).RG}{\frac{RG}{f_P(RG)} + e_N.f_e(RG).(1 - f_f).[\frac{k}{RP_{CE}} + \frac{1}{RP_{BF}}]} \qquad (7)$$

Note that, as expected, in the special case where bug-fixing and current engineering don't need any resources (e.g., by letting RP_{CE} and RP_{BF} go to infinity), the equation 7 is the same as equation 2, as discussed in the text.

To find the maximum FR rate, one needs to take the derivative of FR with respect to RR (for which we need specific f_e and f_p functions) and equate it to zero. Finding the optimum RR (through numerical or analytical solution), we can plug it back into equation 7 to find the optimum level of capacity.