

Dynamics of Concurrent Software Development

Abstract

In a concurrent development process different releases of a software product overlap. Organizations involved in concurrent software development not only experience the dynamics common to single projects, but also face interactions between different releases of their product: they share resources among different stages of different projects, including customer support, they have a common code-base and architecture that carries problems across releases, they use the same capabilities, and their market success in early releases impacts their resources in later ones. Drawing on two case studies we discuss some of the feedback processes central to concurrent software development and build a simple simulation model to analyze the resulting dynamics. This model sheds light on tipping dynamics, the nature of inter-project feedback loops, and alternative resource allocation policies relevant to management of concurrent software development.

Keywords

Software dynamics, concurrent development, reinforcing loop, capability, multiple release

Acknowledgements

We thank Nelson Repenning, John Sterman, Michael Cusumano, David Ford, three anonymous reviewers, and seminar participants at MIT for valuable feedback on the research related to this paper. We thank Jon Bentley and Tarek Abdel-Hamid for helpful comments on an earlier draft. We thank research participants for their generosity with their time and information in supporting this study. Partial financial support for this research was provided by Avaya Inc.

1. Introduction

Despite the significant economic incentives and large amount of research and publication on best practices, many software organizations fail to deliver on their promises while a few achieve outstanding results. In fact significant performance heterogeneity has persisted across different firms for a long time¹ (Møløkken and Jørgensen 2003) and only a small fraction of new firms join the ranks of the most successful. For example, in just a decade, less than one percent of Indian software firms who became active in the software export business have outperformed others to now account for 60 percent of Indian software export revenue (Ethiraj, Kale et al. 2005).

To understand the variability in software development performance, system dynamics has been used to model and simulate software development projects (Abdelhamid and Madnick 1989; Abdelhamid and Madnick 1991; Madachy 2007). Abdelhamid and Madnick (1983) first introduced a feedback-based perspective into analysis of software projects and identified some of the core rework processes. They later expanded this work to include explicitly quality assurance, multiple types of errors, experience (Abdelhamid and Madnick 1991) and learning dynamics (Lin, AbdelHamid et al. 1997). The focus of AbdelHamid and colleagues has been on single project dynamics and the estimation of project size and effort. Weinberg (1994) has qualitatively discussed several feedback processes in software development organizations, with a focus on controlling problematic dynamics. Madachy and colleagues have used system dynamics for tackling different problems in software engineering (Kellner, Madachy et al. 1999). They focused on separately modeling specific software development issues such as Brooks's law, morale and burnout, learning, software reuse, and business value (Madachy 2005; Madachy 2007). However, previous studies tend to focus on individual projects and few inter-project dynamics are discussed. In this study we document and analyze a few of the feedback processes that cut across multiple software projects within an organization. As a result we use the development organization, rather than the project, as the unit of analysis. This perspective allows us to observe some of the multi-project feedback processes that significantly impact success and failure of software organizations.

¹ Bi-annual survey of IT projects by Standish group 1994-2006 shows continued and significant performance heterogeneity: <http://www.standishgroup.com/>

Figure 1 shows the typical software development process in many software organizations. From popular desktop applications to specialized software for corporate use, the majority of software products come in multiple releases that build on each other and introduce successively better or more feature-rich products. Different releases of a software product are developed and introduced in the market as different projects that include some level of overlap in development time (concurrency). Although the exact development process for a project/release depends on specific organizations and how they work, the basic process is relatively similar across organizations. Each project includes different activities for developing requirements based on the features that are planned to be included in the next release of the software, developing the code to implement those requirements, integrating and testing the software product, and product introduction into the market. Once a release is introduced, potential defects not discovered and fixed in the development phase will surface. Depending on their severity, these defects are fixed through ad hoc problem solving for specific customers or development of patches for all customers. We call such corrective activities by the development group current engineering (CE). Finally, these bugs are fixed in the code base for the future releases of the software. We separate CE and bug-fixing here given some differences in their dynamic implications, but they are in practice connected activities. Note that although the figure shows the process neatly divided into a set of separate phases, in fact the activities overlap in ways that are difficult to represent in a simple diagram.

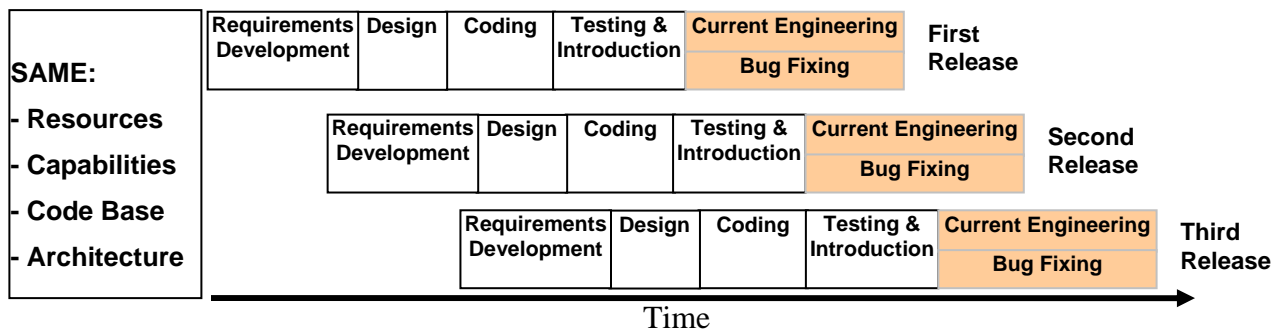


Figure 1-Concurrent software development process for multiple release products. The introduction of a new release happens between “Testing & Introduction” and “Current Engineering” and “Bug fixing” activities. All releases build on and use some of the same elements, highlighted on the left.

What distinguishes this study is the explicit treatment of interconnections among releases: they build on each other and hence they carry over unresolved problems in product design and code. Moreover, resources (primarily people, but also test labs, equipment, and various other items) are shared between development and CE and thus the quality of past releases influences resource availability for the current and future releases in planning and development. Finally, all the work on different releases is conducted using the same set of organizational routines and capabilities. Repenning (2000) was the first to identify the importance of interactions among multiple product development projects. He showed that if product design and testing is given higher priority in resource allocation than concept design, a temporary resource shortage can lead to sustained decline in future project performance, as lower quality of concept design triggers rework in product design and leaves less resources for concept design of the next project, and the cycle continues. Taylor and Ford (2006) demonstrate tipping dynamics in a single project as a result of the impact of schedule pressure on quality. Here we introduce and analyze other coupling mechanisms among different releases of a software product. Specifically, we show how sharing of resources between development and current engineering can lead to dynamics similar to those for concept design and development, and how erosion of development capability and decline in the code base and architecture can contribute to dynamics that drive otherwise similar organizations to face different outcomes. In the next section we discuss the empirical study that motivated the theory building work reported in this paper. In section 3 we discuss a few feedback processes that traverse multiple software releases. Experiments with a simplified simulation model that captures these feedback processes are reported next. Finally, section 5 discusses the contributions and limitations of this study.

2. Empirical background and analysis process

This study is motivated by two detailed case studies in Sigma², a large telecommunications firm. The two software products (Alpha and Beta) were selected following a polar research design (e.g. Eisenhardt 1989). They showed significant differences in their practices and performance outcomes despite being exposed to the same organizational incentives and resource constraints. These cases, provide the empirical background to study dynamics of concurrent software development and how those dynamics can lead to different performance trajectories over time.

2.1. Case Study Overview

Seventy semi-structured interviews (by phone and in person, ranging between 30 to 90 minutes), a few development group meetings, and extensive archival data from Alpha and Beta inform the case studies we draw on. Primary data gathering included 48 initial interviews and gathering of archival data. We gathered additional data for elaboration on different themes after the three months of initial fieldwork. We interviewed members of all functional areas including architects and system engineers, developers, testers, customer service staff, sales support personnel, and marketing personnel, as well as managers on two different organizational levels in several of these areas.

Both Alpha and Beta followed a largely waterfall development approach and engaged in concurrent release development (depicted in Figure 1). Alpha develops a real-time telecommunications product that evolved over 8 years and many releases. It has about 120 full-time employees who work in five main locations. It began as a start-up and was later acquired by Sigma. We focus our data collection on the years Alpha has been in Sigma. Alpha's first few releases led the market and it was considered a promising, strategic product for the company. In the last two years, however, long delays in delivery and low quality have removed it from its leadership position in the market and have cast doubt on its long-term viability.

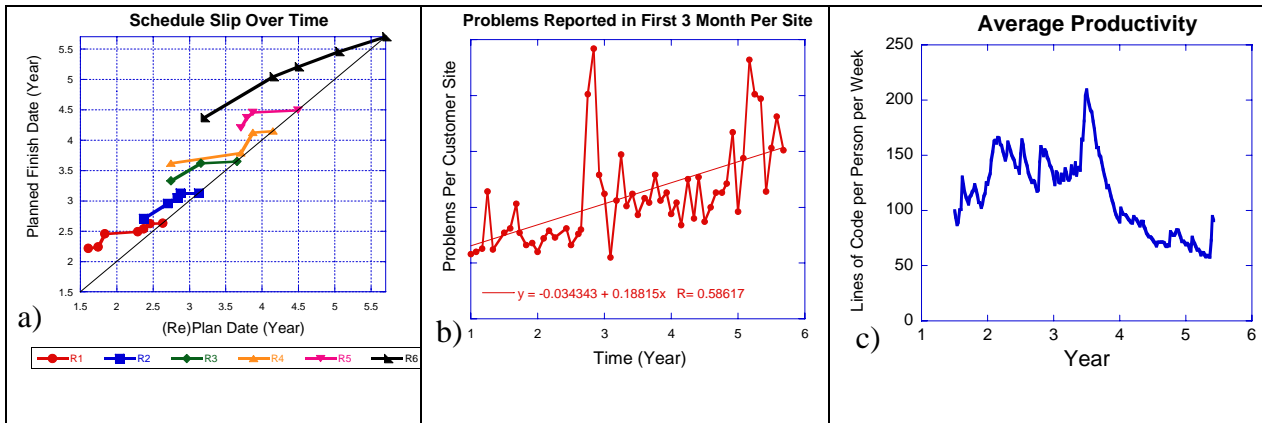


Figure 2- Alpha's historical performance on schedule (a), quality (b), and productivity (c). a) The scheduled finished date (Y-axis) is graphed against the dates of new reschedules (solid data points). A perfect project will have a horizontal line, while the more sloped lines suggest significant delays. b) The average number of bugs per customer in the first three month of installation. The slope of the curve is significantly positive, suggesting a gradual decline in quality. c) The productivity of Alpha team, expressed in the average number of lines of new code added per person per week, shows a significant decline after year three.

Alpha shows a curious mode of behavior: typically organizations learn from their experience and improve their performance over multiple projects (Argote and Epple 1990), yet Alpha illustrates declining performance over time (See Figure 2). It began as a startup that satisfied its customers but over time tended to ignore good development practices in favor of rushing products to market. Review and testing steps early in the development process are compromised under increasing pressure. The development practice departs from the official, espoused, development process guides that call for comprehensive requirements review,

² Pseudonyms are used for organization and products for privacy reasons.

development of detailed requirements, design review, code review, and multiple testing and documentation steps during the development process. An experienced developer in Alpha summarizes this pattern:

“What I have seen is years and years of degradation- getting worse and worse over time”

Growing current engineering load, deteriorating code base, and increasing bug fixes put increasing pressure on the organization. As a result multiple metrics of performance (schedule, quality, productivity) all show little improvement or even decline over the last five years (See Figure 2). The inter-project dynamics that challenged Alpha are at the core of the motivation for this study.

We studied the second case, Beta, to contrast its diverging performance from Alpha, despite similar organizational incentives and resource constraints. Beta, is part of a complex telecommunications switch and is developed largely independently of the parent product but is tested, launched, and sold together with it. Beta includes both software and hardware elements and the majority of its over-80-person R&D resources focus on the software part. Contrary to Alpha, Beta organization has maintained several high-quality development practices including early review and testing, careful interface specification, root-cause analysis, and extensive automation of testing. It has therefore avoided the deleterious effect of previous quality problems hurting the current project and as a result it has maintained low defect rates, avoided delays, and has established itself as a reliable and high quality product.

Our field study departs from a controlled experiment: Alpha and Beta are based on different technologies, originated differently, and have different development teams. Because of these structural differences their observed performance heterogeneity can be partially explained by factors other than the feedback structures common to both cases. Therefore Beta is mainly included to enhance the grounding of feedback loops and simulation model in empirical data and to provide a more nuanced understanding of when different feedbacks are important.

2.2. Analysis Process

The analysis process for this research consisted of three stages: identifying qualitative feedback loops, building a detailed simulation model, and distilling a simple model from the detailed one to summarize the most important insights and communicate those with the broader audience. This paper shares the output of the last stage. However we briefly discuss the overall process, to clarify the connection between the case studies and the models discussed here.

We used an iterative approach to extract different feedback processes from our field data. Interviewees explained different aspects of software development in their organizations and the factors that contribute to performance (e.g. quality and schedule). The interview data were then transformed into potential causal connections as part of a qualitative model. Previous literature provided a theoretical lens for this interpretation and transformation process. Causal links and loops that were confirmed by multiple interviewees were then captured in a detailed simulation model. A similar iterative procedure was used for structural validation (Serman 2000), extraction of table functions (Ford and Serman 1998), and parameterization of this model to replicate the historical behavioral modes (See Figure 2). The detailed model is available elsewhere for interested researchers (Rahmandad 2005).

In the last stage, through sensitivity analysis in the detailed model and consulting the previous studies, we narrowed our focus on the inter-project feedback loops that were both important in the history of our cases and made new contributions to the literature. This subset of feedback loops and the small simulation model that is constructed around them is used in this paper to build theory about reinforcing dynamics in concurrent software development. Given this goal, the simulation model is purposefully much simpler than the two cases or the detailed model. It excludes many factors that are important in fully explaining Alpha and Beta’s evolution, and focuses on how a core set of inter-project feedback processes can lead to tipping dynamics and performance heterogeneity in software development. As such, this paper follows a rich tradition of using case studies and simulation models to build grounded theory (Glaser and Strauss 1973; Davis, Eisenhardt et al. 2007) on dynamic social phenomena (e.g. Forrester 1968; Serman 1985; Ford and Serman 1998; Repenning 2002).

3. Modeling Concurrent Software Development Dynamics

In this section we discuss feedback processes that cut across multiple releases of a software product and that we found to be significant in our case studies. While several reinforcing feedback loops have been identified in project management in general (Lyneis and Ford 2007) and software development in particular (Abdelhamid and Madnick 1991; Madachy and Khoshnevis 1997), the focus on feedback effects that cut across different projects and highlight an organization as the unit of analysis distinguish the contribution of this research.

We model the development of multiple releases of a software product. New features are added to the backlog of *Features Under Development* through *Feature Request* rate. The *Feature Release* rate identifies the introduction of new versions of the software into the market. This rate depends on multiple factors including the number of features under development, the development (and testing) *Human Resources* available, the gross *Productivity* of these resources (in terms of features developed per developer per month), the *Defect Rate*, i.e. quality of work by these resources, and the comprehensiveness and speed of quality control processes used to detect potential defects before release (*QC Quality*). Release introductions increase *Features in Market* and in parallel with feature release, *Defect Generation* introduces more *Defects in Released Code*. In reality feature release often happens in discontinuous batches (releases). However, for simplicity we represent a smooth flow of features into the market in our modeling work.

Several feedback processes were identified from the interview data and observation of the development process in Sigma. In line with previous research, schedule pressure and different organizational reactions to it emerged as a central concept in our analysis (Cooper 1994; Graham 2000; Ford and Sterman 2003; Nepal, Park et al. 2006; Taylor and Ford 2006). Schedule pressure measures the gap between available *Human* (and other) *Resources* required to develop *Features Under Development* on time, fix the bugs in previous releases, and invest in capability development. A project that is short of resources, close to a deadline, or has many features to develop, could have a high schedule pressure. Working harder is one of the typical reactions to an increased schedule pressure. By working harder the development team can increase its gross *Productivity*, measured in, say, features per month per developer, and therefore increase the *Feature Release* rate. That alleviates the pressure through reducing the *Features Under Development*, closing the balancing loop “*Work Harder*” or B1 (Figure 3).

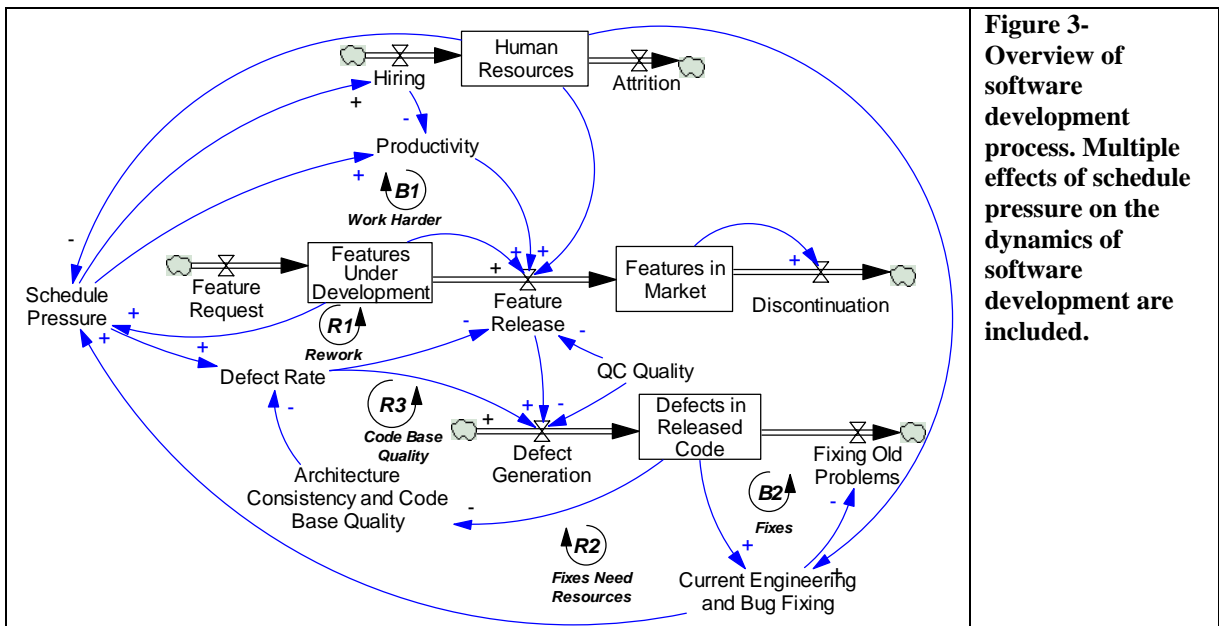


Figure 3- Overview of software development process. Multiple effects of schedule pressure on the dynamics of software development are included.

A potentially troubling side effect of working faster is loss of quality. Under pressure many individuals tend to cut corners and skip beneficial process steps. Such tradeoffs can slow down the long-term feature release rate because they lead to higher *Defect Rate*. Once discovered in the testing process, these

defects should be reworked and therefore slow down the new development activities. This leads to further increase in *schedule pressure* and the reinforcing feedback loop “*R1: Rework*”. Abdel Hamid (Abdelhamid and Madnick 1983; Abdelhamid 1990) first identified the importance of these dynamics for scheduling software development projects. Sigma developers identified some of the specific processes that were compromised under schedule pressure to include detailed requirements development (the step between general requirements developed in design and the coding work), code review (review of code developed by other team members to catch problems), unit testing (testing of specific functionalities a module is supposed to deliver, usually done by developers themselves), and documentation.

Problems that are found in the field need to be fixed through current engineering (CE) and bug fixing. Solving these problems avoids costly customer dissatisfaction and improves the code base for future releases (“*B2: Fixes*”). The bug fixing and CE activities however require scarce *human resources* and can lead to further *schedule pressure*. This in turn increases the defect rate in the *next* release and the future *Defects in Released Code* thus closing another reinforcing loop: “*Fixes Need Resources*” (*R2*). This reinforcing loop is different from the previous one in that it crosses multiple releases of a product: defects in previous releases impact resource availability for current release. Alpha team frequently spent as much as 30% of their resources on current engineering. In contrast, Beta’s high quality allowed the team to limit current engineering to 10% of total resources. If the bugs are not fixed in the code base, the next releases of the software face higher error hazards because old bugs may interact with new features and architectural problems will complicate addition of new capabilities, leading to another reinforcing loop across multiple projects: “*Code Base Quality*” (*R3*). In fact in the last two releases of Alpha the team faced a dilemma: given the low quality of underlying code base and architectural problems, should they scrap the old architecture and code base and start developing the product anew, or try to utilize the old architecture and accommodate new features by ad-hoc solutions. Under schedule pressure the organization opted for the latter option, and thus faced many inefficiencies in the development process.

Figure 4 introduces several multi-project reinforcing loops that relate to market performance of the software product and the organizational capabilities. Project performance impacts the resources available to the software organization. We capture this connection through *Market Success*, which aggregates relevant market performance metrics (e.g. market share, revenue, profitability). For independent organizations, there is a direct link between revenue and available *Financial Resources*: only a fraction of revenue can be invested back in the organization. For organizational units, such as Alpha and Beta, the link is weaker, but still exists. Typically the budget for the organization is a function of its historical performance. Sustained low performance reduces the viability of a product line and cuts its resources in the portfolio of the firm’s investments while star products are in a better position to negotiate expansion of their resources. Indeed Alpha’s initial market success triggered its acquisition by the Sigma organization, and fueled continued investment in the product for several years. However, in Alpha’s case, the resources given to the organization were not cut quickly in the face of later poor market performance. A combination of over-optimistic future performance predictions by Alpha managers and perceived strategic significance of presence in Alpha’s market niche helped sustain resource flow to the organization.

Market success is influenced by project performance in several ways. Products with high levels of *defects in released code* require additional costs for service and maintenance in the market place, thus negatively impacting *market success* and resource availability. Market success is also directly impacted when the product is known for poor quality in the market and sales (or price premium) is compromised as a result. Lower market success in turn limits the available resources and can trigger disruptive pressures, thus closing two reinforcing loops, “*R4: Life Cycle Costs*” and “*R5: No Money*”.

Another important concept in our analysis is the organization’s *software development capability*. Development capability represents the routines and elements that allow the development organization to produce high quality software. It consists of many factors, including, but not limited to, the skills of the individual members, the robustness of the development and testing processes used, the availability of automated testing capabilities, and the accessibility and appropriateness of the technology used by the team. Development capability changes relatively slowly but when it does, its impact on *productivity* and *defect rate* can be significant. It therefore has an important indirect impact on *market success* and *schedule pressure* in the company. A large body of research in strategy has identified product development as one of the core

capabilities essential for success of firms in dynamic markets (Eisenhardt and Tabrizi 1995; Teece, Pisano et al. 1997).

Development capability can be increased through training, process improvement, and tool development inside the organization. Other ways include experiential learning in communities of practice and learning from examples of other successful groups. Activities contributing to capability building are often compromised under schedule pressure, or in the absence of financial resources, leading to the reinforcing loop “R7: *Capability Building*”. Development capability increases productivity and quality of development activities by establishing robust processes and providing tools that facilitate software development. Faster and error free development allows for lower schedule pressure and better financial performance, thereby freeing up resources to be spent on building the capability further (Crosby 1996). This reinforcing loop is identified as “R8: *Robust Processes*”. Product development capability also helps develop products with better maintainability, compatibility, configurability, serviceability, and installability. These characteristics enhance the attractiveness of the product in the market and lead to better sales, “R6: *Peripheral Capabilities*”. Alpha and Beta differed significantly in their development capability. Beta had successfully integrated many automated elements in their processes and actively encouraged team members to build new tools that could improve the quality and productivity of the group. Under sustained schedule pressure, little capability building activity was conducted in Alpha on a regular basis.

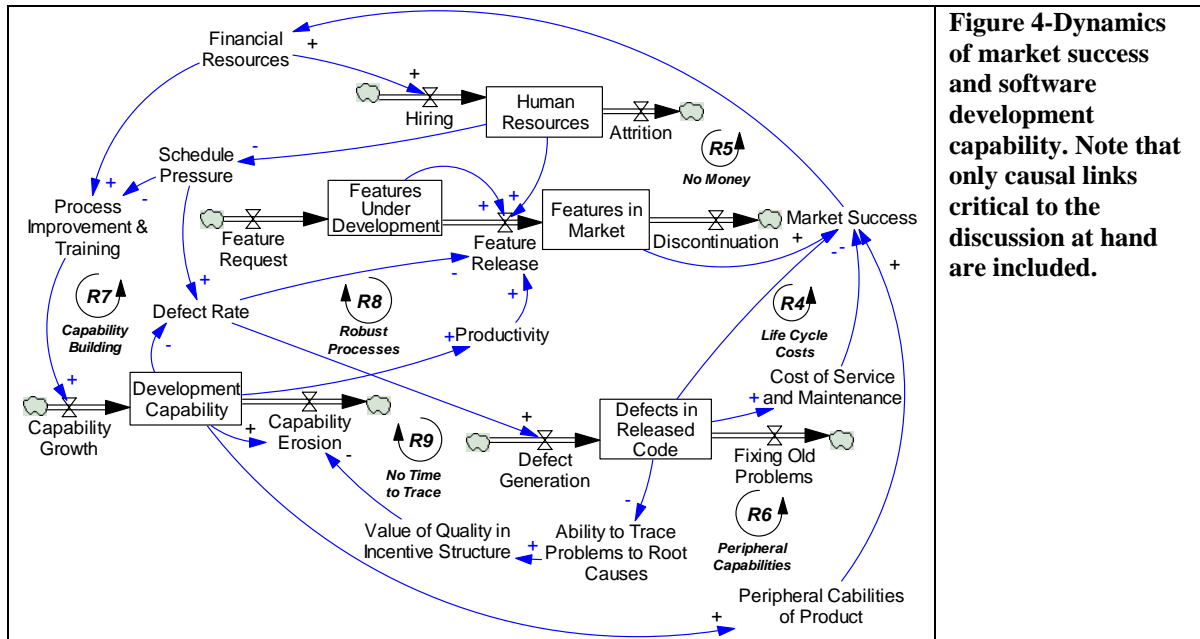
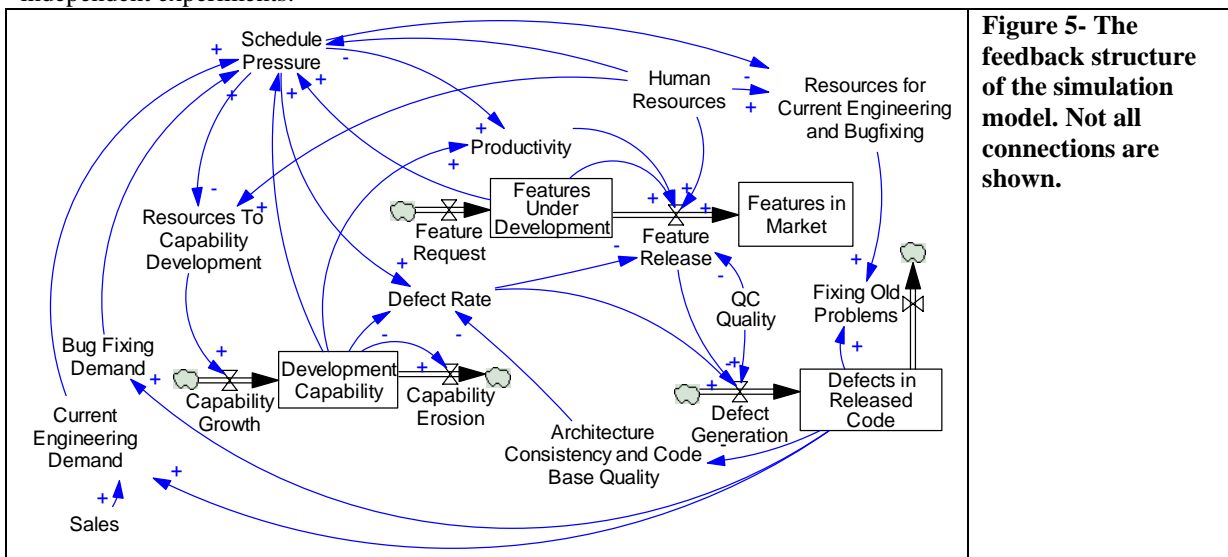


Figure 4-Dynamics of market success and software development capability. Note that only causal links critical to the discussion at hand are included.

Finally, a subtle but important organizational impact of having many *defects in released code* is on the incentive structures in the organization. High quality organizations can drill down to the root causes of the few bugs they find after the release of the product, distinguish the problematic processes and individual practices, and remedy these root causes to avoid future recurrence. This was indeed actively pursued in Beta. On the other hand, product groups overwhelmed by many quality issues have little opportunity to find the root causes and tie individual incentives to the quality of their work. In fact, in such organizations fire-fighters, the members who save the customers in trouble, are often recognized to the detriment of those who strive to produce a high quality product in the first place. Such organizational structures systematically reward practices that ignore quality in original development and shift the resources towards fixing the problems once the product is released and the damage is largely done (“R9: *No Time To Trace*”).

4. Simulating dynamics of concurrent software development

In this section we analyze, through simulation experiments, the implications of some of the qualitative feedback processes developed in the preceding. We use a stylized simulation model that focuses on a subset of the inter-project feedback loops to find out if, and how, these loops can induce diverging performance across organizations involved in concurrent software development (Figure 5). This model therefore leaves out many of the nuances in our cases: it assumes fixed *human resources* and *feature requests*, uses generic parameter values not calibrated to either case, and assumes constant sales. Therefore the main dynamics in the simulations following arise from the tradeoffs in allocating limited organizational resources among new development, capability building, current engineering, and bug fixing. In the base case we assumed these activities have equal priority in the allocation of resources. Desired resources for capability investment are assumed to replenish erosion of capabilities and allow for a moderate growth. The balance of desired vs. available resources determines the schedule pressure, which in turn impacts the rates of change in different stock variables. **Figure 5** represents the main feedback processes we capture in the simulation model. The full model with documentation is available in supplementary material for download and independent experiments.



In the following experiments we simulate an organization that is originally in dynamic equilibrium, that is, the total resource requirements for new development, capability growth, bug fixing, and current engineering is balanced with the available resources and therefore the *Features Under Development*, *Defects in Released Code*, and *Development Capability* are roughly constant³. We then expose the organization to a temporary increase in *Feature Request*, raising the demand for new features by ten units for a period of one month, before returning to the initial *feature request* of five. The reaction of the simulated organization to this input informs the stability of the organization and the dynamics of its capability and performance. We report the performances of three variables that illustrate different characteristics of the organization. *Feature release* is used as a proxy for the performance of the development organization. It reports how many features per month are released to the market by the development organization. *Development capability* reports on the quality and robustness of the underlying development practices. Finally, we define *Total Costs of Recovery* as the person-months of resources needed to take the organization back to its original equilibrium. This measure includes the costs of developing the pending features (to catch up with market demand), fix the

³ The structure of this nonlinear system does not include a non-trivial equilibrium, but behavior very close to equilibrium is attainable for time horizons of interest.

problems in the released code, and take the capability back into its original state. Conservatively, we measure these costs assuming base level quality and productivity⁴.

4.1. Firefighting in concurrent software development

Figure 6 shows the base case performance of the simulated organization. The graph shows four different variables in three separate graphs, where *feature request* (curve labeled 2) is replicated in all of the graphs. Note that we are showing feature release continuously, despite real market introduction usually happening in batches (in our cases about 30-50 features per release, taking ~ 6-10 months to complete). The short-term surge triggers a surprising chain of events. The increase in *features under development* moves resource allocation towards development activities, to the detriment of investment in the capabilities. While a short-term increase in *feature release* is achieved because of higher demand and increased productivity under schedule pressure (Loop B1 in Figure 3), that advantage gradually fades as the side effects of increased defect rate (Loop R1 in Figure 3) and lower capability investment (Loops R7-8 in Figure 4) take effect by the end of the first year after the shock. The organization finds itself to be working harder, sacrificing good practices (loop R1), cutting down on competency development (loop R7), and yet (thus) achieving lower feature release rates (Figure 6-a). The lower feature release further increases the pressure (because the features under development keep accumulating) and erodes capability (Figure 6-b). These dynamics continue to plague the organization several projects later as the reinforcing loops (including R3, Code Base Quality) completely take over and push the organization to a new mode of development where low productivity, low quality, and high costs are embedded in software development practices. The cost of recovering the organization under these conditions grows fast once the reinforcing loops take over (Figure 6-c).

These dynamics represent the tipping mode of behavior that can dominate an organization under strong reinforcing loops (Repenning, Goncalves et al. 2001). Here a relatively small exogenous change can have a significant impact on the organization and take it to a qualitatively different region of behavior. Following Repenning (2001) we call this mode of behavior firefighting, in which people are spending most of their time on emergency problem solving, projects are late, the team is under significant pressure, and despite working hard the situation does not improve. These dynamics unfold over relatively long time horizons (years), and thus cross many development projects. This fact highlights the importance of taking the development organization as the unit of analysis, as actions potentially effective in one project (spending more resources on development activity) can prove counter productive for the organization over the long term (e.g. through the erosion of development capability or degradation of code base).

The base case results also reveal how the dynamic hypotheses captured in this model can lead similar organizations to completely different performance outcomes. An unexpected surge in feature requests or current engineering demand, as well as temporary cuts to resources available can tip an otherwise healthy organization into the slippery slopes of spending less time on capability building or good development practices, and getting into more trouble over the coming releases. Once the organization falls into this mode of behavior, however, the costs of recovery continue to grow and the prospects of a turnaround diminish. This was indeed the daily experience of Alpha towards the end of this study: the product was falling behind the market, resources were limited and likely to shrink because of poor performance, and development capability was low. This left the organization with few viable options to get back on track.

Next (sections 4.2- 4.4) we consider three aspects of these dynamics: 1) How do the reinforcing processes impact organizational resilience against tipping dynamics? 2) What are the similarities and differences between feedback effects that depend on schedule pressure and those depending on capability development? 3) How do alternative resource allocation policies impact the dynamics?

⁴ Productivity and quality are likely to be less than base levels for an organization recovering from poor performance (given the eroded capability), therefore using base levels gives conservative estimates.

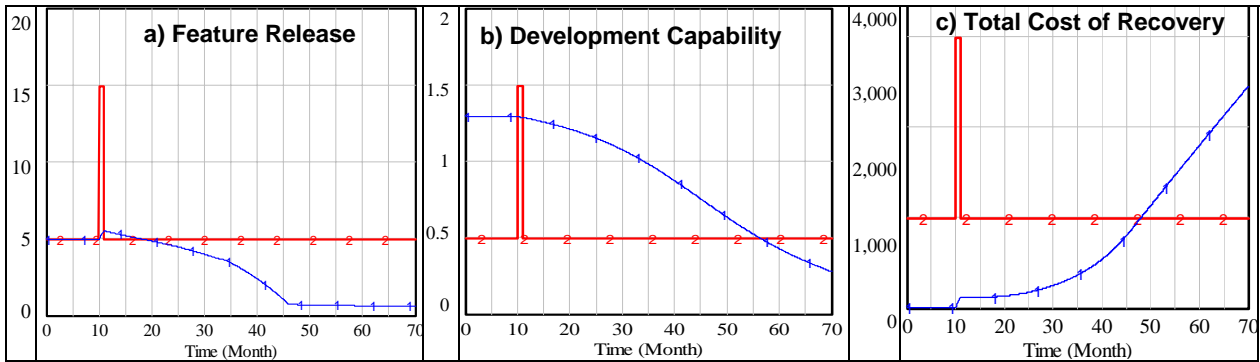
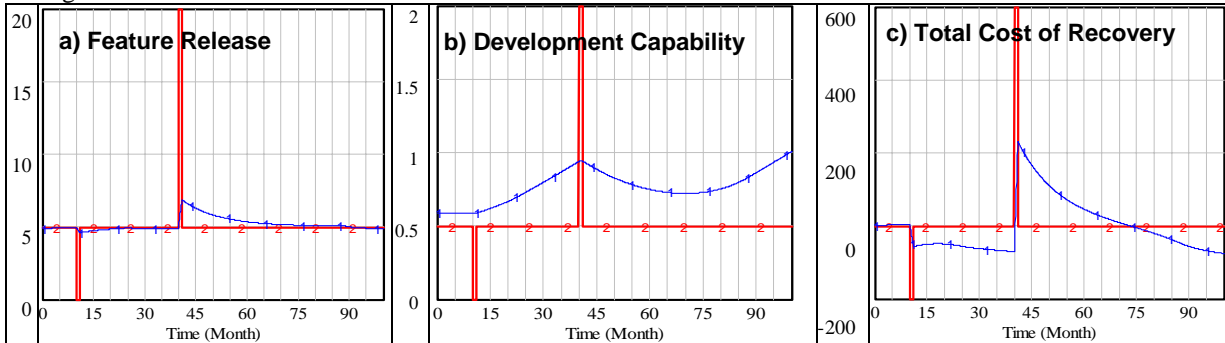


Figure 6- Base case dynamics. The three metrics of performance are reported along with the pulse feature demand input to the system (follows the scale in graph a, repeated in all graphs). The Y axes are a) Feature Release in Feature/Month b) “Development Capability” which is Dimensionless c) “Total Costs of Recovery” expressed in Month*Person. Feature release saturates at a low level, but total cost of recovery (and features under development) continue to grow.

4.2. Virtuous cycles and organizational resilience

The impact of a reinforcing loop can be either vicious or virtuous depending on the direction of its activity. Previous experiment highlighted vicious cycles. A reinforcing loop active in the virtuous direction can lead to the accumulation of development capability, reduction in the problems in the released code, and therefore an increase in the potential capability of the organization to create new features. An organization that has benefited from virtuous loops becomes more resilient, that is, it can tolerate much stronger external pressures (e.g. additional demand, lack of resources). Figure 7 shows one such experiment. At time ten the demand is reduced to zero for one month, to give the organization the slack needed to build its capabilities⁵. The virtuous capability building loops are slow, but they have their impact. After 2.5 years, the organization is tested with a significant, temporary increase in demand (demand going from 5 to 30). Despite the very large shock, which would have devastated the original organization (See Figure 6 where an increase to 15 features tipped the system into firefighting) the organization survives and returns towards its capability building mode of operation. As a result of initial slack provided, the reinforcing capability loops are activated in the virtuous direction between the months 10 and 40 and lead to increasing levels of development capability (Figure 7 -b). This cumulative capability acts as a safety cushion that comes to the rescue of the organization over the next, potentially catastrophic disruption. The next pulse (Month 40, in Figure 7) triggers a long period of high pressure (significant increase in features under development), and reduced investment in capability. Nevertheless the organization survives because it uses its high levels of capability manifested in high quality and productivity levels embedded in individual knowledge, organizational practices, and development tools. Capability is hit during this period (the decline in item b from month 40 to 70), but it remains high enough to overcome the challenge by higher feature release (item a) and get the organization back on track.



⁵ Alternatively we could increase available resources temporarily and get very similar results.

Figure 7- Increased resilience after activation of virtuous loops. Items marked with 2 are the feature request rate. The organization is given a one-month break at month 10 (zero feature request) to start building capabilities (b). Later, at time 40, a strong pulse of magnitude 25 feature request tests the organizational resilience (feature request’s scale is same as graph *a* in all three figures, goes over the scale at time 40). The organization recovers from this strong shock (can retain its feature release (a) at/beyond steady-state feature request). Features under development and total costs of recovery (c) never get out of control.

4.3. Schedule pressure and capability impacts

The majority of reinforcing feedbacks in our simulation experiments go through schedule pressure or capability development’s impacts on productivity and quality. Schedule pressure feedbacks are faster and thus can unfold within a single project while the capability dynamics usually take more time to emerge given the inertia of the capability stock. Therefore capability related feedback processes often work across several projects. We compare the impact of these two different influence pathways in two simulation experiments (See Figure 8). In the first experiment we remove the impact of schedule pressure on productivity and quality, therefore removing feedback loops R1, B1, and R2 (Figure 8-a). To make the simulations directly comparable with the base case, we fix the impact of schedule pressure on quality and productivity at their equilibrium, base case, levels. In the second experiment, we remove the impact of capability dynamics by fixing the value of capability stock at its base case level and removing the resources allocated to capability development from the pool of available resources (Figure 8-b). These changes effectively cut the loops R7 and R8 (in Figure 4). This experiment resembles an organization that has institutionalized uncompromising capability building practices such as training, tool building, process improvement, and employee retention programs, and does not allow these processes to be compromised under resource pressures. Figure 8 reports the three performance metrics for these two experiments.

In both cases the tipping dynamics persist despite the removal of the other factor. On the other hand, the extent of impact of the two mechanisms is qualitatively different. Schedule pressure changes relatively fast with organizational conditions, and its impact on error rate and productivity can saturate quickly. Once that saturation point is reached, the organization converges to a new plateau in its performance which may not be too far from initial performance levels (e.g. see feature release in top left, Figure 8). Costs of recovery for the organization under these conditions are also mild compared to the base case. In contrast, feedback processes that go through the capabilities can continue to erode quality and productivity for a long time, even if schedule pressure has no direct impact on quality. These dynamics lead to significant long-term costs and a gap between actual and potential performance. As long as resource pressures cut down investments in capability building, the capability stock declines and leads to increasingly lower levels of productivity and quality. These in turn contribute to further delays and schedule pressure. In effect, productivity and quality are anchored in the current capability of the firm. This anchor determines the range of change for productivity and quality under different levels of pressure. Sustained resource pressure, however, can erode the capability, shifting that anchor continuously, in a process similar to erosion of quality norms in service operations (Oliva and Sterman 2001). This process can lead to the institutionalization of poor development practices in an organization. In fact some of the older Alpha employees pointed out that given the current, inefficient development processes, new employees do not have an opportunity to observe more effective procedures for software development and therefore cannot internalize some of the basic capabilities and skills.

	Feature Release	Development Capability	Total Cost of Recovery
--	------------------------	-------------------------------	-------------------------------

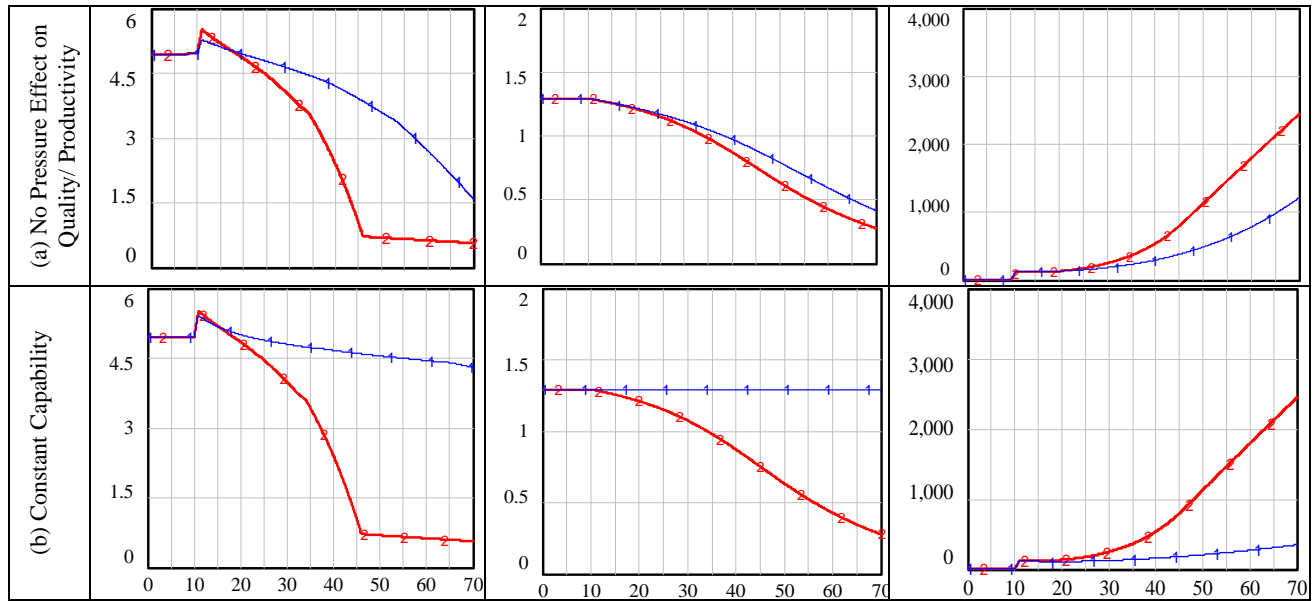


Figure 8- Impact of schedule pressure and capability feedbacks. Feature release (left), development capability (middle) and total cost of recovery (right) are compared with the base case from Figure 6 (marked with 2) for two experiments where a) impact of schedule pressure is removed at time 10 (top row) and b) when capability dynamics are excluded (bottom row).

These experiments further highlight the significance of taking the software development organization as the unit of analysis. Because capabilities reside in organizational routines and practices that change only slowly and in the span of multiple development projects, their impact on performance of software projects is also best seen across multiple projects. While effect of schedule pressure on quality is more salient and influential within a single project, the capability dynamics tie together multiple projects in a reinforcing cycle of capability-performance-resource demand for bug fixing and development-and resource availability for capability investment. The long-term impact of this feedback process on tipping dynamics in software development organizations can be quite substantial.

4.4. Impact of alternative resource allocation policies

In the analysis so far, we have assumed that investment in development, current engineering, capability building, and bug fixing have similar priorities, thus resource pressure impacted these activities similarly. This assumption is helpful to understand the basic dynamics without complications arising from alternative priority settings. However, in practice organizations often have different priorities for different activities. For example Alpha had a fairly high priority for current engineering activities because customers were directly affected by current engineering. They demanded quick reaction from the Alpha team to fix the problems in the field as soon as those problems were observed. In fact Alpha often used resources otherwise devoted to development for taking care of surges in current engineering demand. Beta, in contrast, gave higher priority to development and capability building.

Alternative allocation policies also inform managerial interventions aimed at reducing risks and costs of tipping dynamics. We therefore conduct a set of experiments in which we change the priority of different investment options at month 10, when the external shock is introduced, and track the resulting performance of the simulated organization. For this purpose we increase the priority of each activity, i.e. development, current engineering, capability building, and bug fixing, in a separate experiment. In each experiment the activity with the high priority will be first in line for receiving the resources it desires. The three remaining activities will then share the remaining resources proportional to their demand.

	Feature Release	Development Capability	Total Cost of Recovery
--	-----------------	------------------------	------------------------

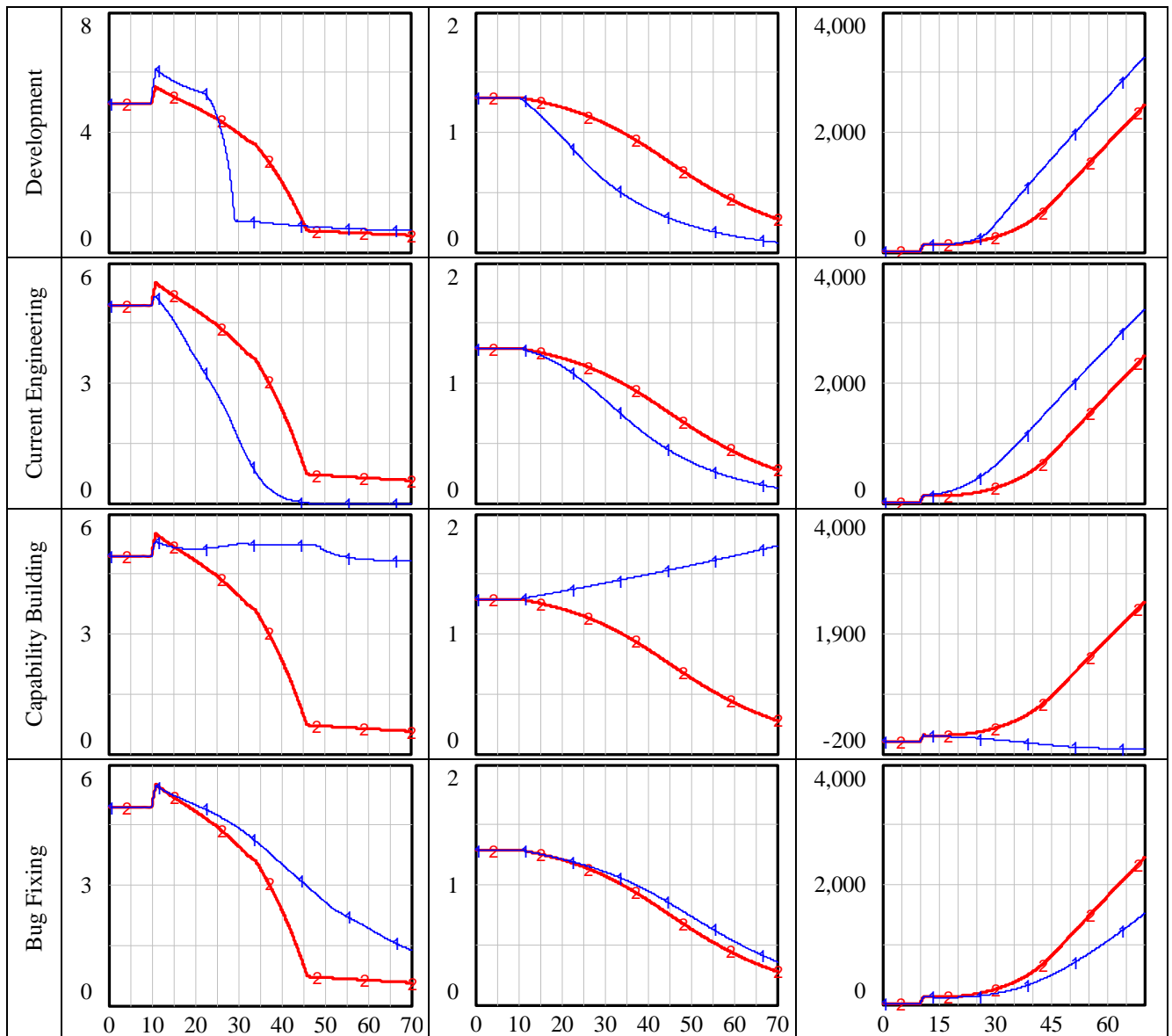


Figure 9- Alternative priority setting policies. Left column indicates the activity that has been assigned the highest priority in the simulation results reported in that row. Three metrics of performance, feature release (left), development capability (middle), and total cost of recovery (right) are reported for the base case (item marked with 2) against the alterantive priority level case (item marked with 1). The performance for the first few months after the shock is of special importance because it contains the most salient information cues managers receive.

Figure 9 reports the results of these experiments. A few characteristics of these results inform our discussion. First, giving high priority to development (first row in Figure 9) is most successful in increasing short-term feature release, e.g. rescuing the current release from delays. However, by ignoring the development capability, the organization may still be pushed into the firefighting mode of behavior and in fact face lower long-term performance than the base case. Overall this can lead to significant costs for the organization. Not surprisingly, raising the priority of current engineering (second row) does not help any of the performance metrics we track. It hurts both the feature release and the capability and therefore leads to more costs than the base case. It should however be noted that we are not tracking customer satisfaction and

loyalty, which are the main outcomes benefiting from higher priority of current engineering. The impact of those factors on sales and therefore market success may somewhat raise the value of current engineering. Nevertheless, as long as development capability declines because of resource pressures, the organization will tip into the firefighting mode of behavior and will under-performs its potential. Raising the priority of capability development beyond other activities (third row) promises the best long-term outcomes. The tipping dynamics are avoided, at the cost of the short-term decline in feature release compared to the base case. The reason is that by allocating all the desired resource for the capability development activities, the organization grows its capability and thus its productivity and quality. While the schedule pressure induced by the temporary shock at time 10 lowers quality and thus feature release, over longer times the increased capability more than compensates for that loss. Further analysis shows that this allocation policy immunizes the organization for much larger shocks as well. However, very large shocks (e.g. 100 new features, rather than the 10 used in these experiments) will still force the organization into firefighting, as the short-term reduction in quality will lead to growing level of defects in the code base that will activate the *Code Base Quality* (R3) loop. Once the integrity of underlying architecture and code-base is compromised, the organization faces an uphill battle in building new products on a shaky foundation. The long-term impact of R3 is further highlighted by the last experiment where bug fixing is given the highest priority. In this case, even though the allocation policy does not stop tipping, it can reduce its impact by keeping the code base from accumulating too many defects and therefore keeping the defect rate under control because of underlying code base and product design.

These results raise an empirical dilemma: if capability development merits higher priority, why in practice do many organizations (such as Alpha) not follow this allocation policy? One answer may lie in the alignment of capability development with general firm strategy. Whereas the operational excellence that comes with capability development is the strategic thrust for some firms, others may put higher emphasis on innovation or customer intimacy (Rifkin 2001) and thus sacrifice quality, cost, or time to market for other goals. Our case studies suggest that at least two other mechanisms contribute. First, the incentive structures in Alpha (and many software organizations in general) are such that current engineering gets a high priority. Customers in need of help are vocal and visible to the higher management. Developers who save a customer in crisis gain favor with higher management which is sensitive to customer input. Moreover, the quality of work is hard to detect when the development work is being done, and therefore it is hard to reward developers for their high quality work in the first place. Also, with a growing number of problems in the field, the opportunity to trace the root causes of problems and assign responsibility to individual developers shrinks, exacerbating the incentive problem through the *No Time to Trace* (R9) loop. Second, the speed and salience of feedback received from each activity can introduce a bias against less salient or more delayed activities. Current engineering brings quick and clear rewards (customer satisfaction, management attention), development brings reward with some delay (when the project is successfully completed), bug fixing and capability development have even longer and more obscure connections between resource investment and observation of results. Research has shown that people tend to learn the wrong lessons in presence of delays and feedback complexities, often to the detriment of long-term options (Sterman 1994; Repenning and Sterman 2002; Rahmandad 2008).

5. Discussion

Allocation of scarce organizational resources is at the core of management (Mahoney and Pandian 1992). In this study we draw on two case studies to examine a variety of feedback processes that drive resource allocation tradeoffs in many software development organizations. Our analysis shows how several reinforcing loops that cut across multiple projects can drive a wedge between successful and unsuccessful development groups. Next we specify important boundary assumptions and limitations of the study followed by discussion of the contributions.

5.1. Boundary Assumptions and Limitations

The empirical base for this research comes from only two case studies. Even these two cases differed in their technological factors, managerial experience, and initial conditions. Therefore their diverging performance over time may be partially explained by factors other than the tipping dynamics discussed here. However, our analysis establishes that even in the absence of such structural differences, the dynamics resulting from capability erosion or schedule pressure effects, through inter-project feedbacks, can tip

otherwise identical organizations into significantly different performance domains. In this section we discuss some of the main assumptions in our analysis and how they impact the application of our results.

Development Process- The overlapping release schedule increases the interaction between different releases and the tradeoffs between different activities of development, e.g. requirements planning, development, and current engineering of different projects. Therefore the feedback loops that relate to tradeoffs across different activities of the development process (e.g. R3-Fixes Need Resources) are most salient at higher levels of concurrency in the development process. Moreover, the groups we studied largely followed a waterfall development approach. While most of the dynamics discussed in this paper apply to the continuum of waterfall to iterative development processes, the definition of “Development Capability” is largely contingent on the development process at hand, and operational advice should be tailored accordingly.

Product Characteristics- We studied products with multiple releases developed in a single organization. A growing code-base exists for such products that impact new development. Moreover, the market success of previous releases impact the performance and resource availability for the future releases. Organizations that work on a single-release product do not face the feedback processes that are dependent on the multiple-release characteristic or the underlying code-base. Also the value of quality and therefore the importance of investing in capabilities, current engineering, or development, depends on the nature of product. Mission critical products such as communication switches demand much higher quality levels throughout the development process, which reduces the impact of feedback loops related to problems in the field and underlying code base.

Market Characteristics- Our cases come from relatively stable and mature markets where competitors compete over efficient development practices and small efficiency gains are critical to success. However, emerging markets often call for different approaches. Defining customer requirements becomes critical, innovation may be central, and fast and aggressive response benefits early movers through multiple learning and network effects (Majd and Pindyck 1989; Katz and Shapiro 1994; Fudenberg and Tirole 2000). These additional feedback processes may alter some of the policy conclusions and actions indicated when extending results to emerging markets. Similarly, start-up firms have very limited cash and their main goal is to reach the release of the product in the market before the seed money runs out. Capability development is often not highest on their priority list and they may correctly prefer to sacrifice many of the long-term dynamics in order to survive until the revenue stream starts to flow (Rahmandad 2007).

Some other limitations should be noted in the application of these results. The model presented here is very simple with generic parameter values. Empirical estimation of the strength and speed of different feedback processes is an important extension of this work. While detailed time series data required for such estimation is hard to collect, the theoretical and operational contributions of such studies are also great. Additional case studies and large sample empirical analyses are needed to add to the list of important feedback loops and to extend the results into other contexts. More detailed and realistic models can also underlie simulation-based learning environments where managers are put in charge of a simulated software organization and learn from that experience about better real-world management practices (e.g. Sengupta and AbdelHamid 1996).

5.1. Contributions

We identified three major ways through which multiple software development projects interact with each other in reinforcing processes. First, multiple releases of a software product are interconnected because they share resources for development, current engineering, and bug fixing. Low quality of a previous release can put pressure on development of a current release because it diverts resources to current engineering and bug fixing. Moreover, different releases are connected through the accumulating code base and product design. Quality of old releases thus impacts the quality of current work. Second, by considering the impact of market performance on availability of organizational resources, we close several reinforcing loops of the “rich gets richer” nature (R4-R6). Finally, the investment and changes in software development capability in the organization is a function of the performance of different projects, and connects them through a common set of routines and practices used across those projects.

These findings complement the existing literature by focusing on dynamics relevant to software development organizations with multiple projects. Through simulation experiments, we showed how these multi-project feedback processes lead to tipping dynamics that can push the software organization into an

inefficient mode of working harder and achieving less, eroding development capability and market performance (Repenning 2001; Taylor and Ford 2006). We introduce new feedback processes to this literature and analyze the impact of alternative resource allocation policies. Understanding tipping dynamics is important because in them lies the critical challenges and opportunities in managing an organization. Where these dynamics exist, small interventions can have disproportionate impacts, and lead to qualitative differences across organizations. Strong reinforcing loops are therefore associated with strategic opportunities and challenges in different markets (Achi, Doman et al. 1995) such as path dependencies (Arthur 1989), network and complementary goods effects (Cusumano, Mylonadis et al. 1992), and learning curves (Argote and Epple 1990). Firms that understand these dynamics and build their strategies around benefiting from reinforcing loops and avoiding their negative consequences can gain and sustain strategic advantage over their competitors. Beyond theory building research that identifies important reinforcing loops, system dynamics can contribute to this literature by a) quantifying the strength of different feedback processes through quantitative parameter estimation and b) designing and testing control policies that monitor the right variables with measures sensitive to organizational politics and offer appropriate managerial response contingent on the state of an organization.

Several practical implications follow. Given these tipping dynamics, the core managerial challenge is to avoid, under pressure, capability erosion and quality sacrifice that tips the software organization into fire-fighting. Three broad strategies can support this goal. First, priorities should be set to put enough emphasis on capability building and core product development activities. Priority setting should go beyond management mandate and should tie individual incentives to the quality of their work and their investment in capability building e.g. through root cause analysis, dedicated learning time, and social sanctions against poor quality in code delivered (Cusumano and Selby 1995). Furthermore, it is important to identify and schedule capability building projects to be undertaken at the times of relative low work-load, which happen frequently between project peaks. It is also important to seal-off the development team from customer pressure to avoid a bias in favor of (otherwise inefficient) current engineering and bug fixing activities. This should be designed carefully into the process because on the other hand communication with customers is critical for developers to better understand customer requirements in up-stream design and development phases.

Second, the current levels of work pressure and work quality should be monitored on the ground to provide timely signals to management if the organization moves towards firefighting. The metrics monitored for this purpose should be different from those used in the performance review process, so that honest communication of potential problems is not discouraged. In general enhancing communication between management and the development teams is critical for timely identification of, and reaction to, possible slips. Moreover, projects often face surprises that can lead to tipping dynamics; planning explicit buffer times in the project schedule is important to control this risk.

Finally, our analysis suggests that taking more work than its capacity is a significant contributor to an organization's risk of falling into firefighting dynamics. Some possible recommendations to avoid this imbalance include using better estimation processes to match workload with organizational capacity and establishing stringent criteria for starting new projects. The latter issue is important because given commitment escalation and sunk cost dynamics it is behaviorally much easier not to start a project than to stop one later when the organization finds itself unable to handle the workload (Schmidt and Calantone 1998).

References:

- Abdelhamid, T. and S. E. Madnick (1991). Software project dynamics: An integrated approach. Englewood Cliffs, NJ, Prentice-Hall.
- Abdelhamid, T. K. (1990). "Investigating the Cost Schedule Trade-Off in Software-Development." Ieee Software 7(1): 97-105.
- Abdelhamid, T. K. and S. E. Madnick (1983). "The Dynamics of Software Project Scheduling." Communications of the Acm 26(5): 340-346.
- Abdelhamid, T. K. and S. E. Madnick (1989). "Lessons Learned from Modeling the Dynamics of Software-Development." Communications of the Acm 32(12): 1426-1455.

- Achi, Z., A. Doman, O. Sibony, J. Sinha and S. Witt (1995). "The paradox of fast growth tigers." McKinsey Quarterly(3): 4-17.
- Argote, L. and D. Epple (1990). "Learning-Curves in Manufacturing." Science **247**(4945): 920-924.
- Arthur, W. B. (1989). "Competing Technologies, Increasing Returns, and Lock In by Historical Events." Economic Journal **99**(1): 116-131.
- Cooper, K. G. (1994). "The \$2,000 hour: how managers influence project performance through the rework cycle." Project Management Journal **15**(1): 11-124.
- Crosby, P. B. (1996). Quality is still free : making quality certain in uncertain times. New York, McGraw-Hill.
- Cusumano, M. A., Y. Mylonadis and R. S. Rosenbloom (1992). "Strategic Maneuvering and Mass-Market Dynamics - the Triumph of Vhs over Beta." Business History Review **66**(1): 51-94.
- Cusumano, M. A. and R. W. Selby (1995). Microsoft secrets : how the world's most powerful software company creates technology, shapes markets, and manages people. New York, Free Press.
- Davis, J. P., K. M. Eisenhardt and C. B. Bingham (2007). "Developing theory through simulation methods." Academy of Management Review **32**(2): 480-499.
- Eisenhardt, K. M. (1989). "Building theories from case study research." Academy of Management Review **14**(4): 532-550.
- Eisenhardt, K. M. and B. N. Tabrizi (1995). "Accelerating Adaptive Processes - Product Innovation in the Global Computer Industry." Administrative Science Quarterly **40**(1): 84-110.
- 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.
- Ford, D. N. and J. D. Sterman (1998). "Dynamic modeling of product development processes." System Dynamics Review **14**(1): 31-68.
- Ford, D. N. and J. D. Sterman (1998). "Expert knowledge elicitation to improve formal and mental models." System Dynamics Review **14**(4): 309-340.
- Ford, D. N. and J. D. Sterman (2003). "The Liar's Club: Concealing rework in concurrent development." Concurrent Engineering-Research and Applications **11**(3): 211-219.
- Forrester, J. W. (1968). "Market Growth as Influenced by Capital Investment." Industrial Management Review **9**(2): 83-105.
- Fudenberg, D. and J. Tirole (2000). "Pricing a network good to deter entry." Journal of Industrial Economics **48**(4): 373-390.
- Glaser, B. G. and A. L. Strauss (1973). The discovery of grounded theory : strategies for qualitative research. Chicago, Aldine Pub. Co.
- Graham, A. (2000). "Beyond PM 101: lessons for managing large development problems." Project Management Journal **31**(4): 7-18.
- Katz, M. L. and C. Shapiro (1994). "Systems Competition and Network Effects." Journal of Economic Perspectives **8**(2): 93-115.
- Kellner, M. I., R. J. Madachy and D. M. Raffo (1999). "Software process simulation modeling: Why? What? How?" Journal of Systems and Software **46**(2-3): 91-105.

- Lin, C. Y., T. AbdelHamid and J. S. Sherif (1997). "Software-Engineering Process Simulation model (SEPS)." Journal of Systems and Software **38**(3): 263-277.
- Lyneis, J. M. and D. N. Ford (2007). "System dynamics applied to project management: a survey, assessment, and directions for future research." System Dynamics Review **23**(2-3): 157-189.
- Madachy, R. (2005). "Integrated modeling of business value and software processes." Unifying the Software Process Spectrum **3840**: 389-402.
- Madachy, R. (2007). Software Process Dynamics. Forthcoming, Wiley-IEEE Press.
- Madachy, R. and B. Khoshnevis (1997). "Dynamic simulation modeling of an inspection-based software lifecycle process." Simulation **69**(1): 35-47.
- Mahoney, J. T. and J. R. Pandian (1992). "The Resource-Based View within the Conversation of Strategic Management." Strategic Management Journal **13**(5): 363-380.
- Majd, S. and R. S. Pindyck (1989). "The Learning-Curve and Optimal Production under Uncertainty." Rand Journal of Economics **20**(3): 331-343.
- Moløkken, K. and M. Jørgensen (2003). A review of surveys on software effort estimation. IEEE International Symposium on Empirical Software Engineering, Rome, Italy.
- Nepal, M. P., M. Park and B. Son (2006). "Effects of schedule pressure on construction performance." Journal of Construction Engineering and Management-Asce **132**(2): 182-188.
- Oliva, R. and J. D. Sterman (2001). "Cutting corners and working overtime: Quality erosion in the service industry." Management Science **47**(7): 894-914.
- Rahmandad, H. (2005). Three essays on modeling dynamic organizational processes. Sloan School of Management. Cambridge, Massachusetts Institute of Technology. **Ph.D.**
- Rahmandad, H. (2007). Why myopic policies persist? Impact of growth opportunities and competition. International System Dynamics Conference. Boston, System Dynamics Society.
- Rahmandad, H. (2008). "Effect of delays on complexity of organizational learning." Management Science **54**(7): 1297-1312.
- Repenning, N. P. (2000). "A dynamic model of resource allocation in multi-project research and development systems." System Dynamics Review **16**(3): 173-212.
- Repenning, N. P. (2001). "Understanding fire fighting in new product development." The Journal of Product Innovation Management **18**: 285-300.
- Repenning, N. P. (2002). "A simulation-based approach to understanding the dynamics of innovation implementation." Organization Science **13**(2): 109-127.
- Repenning, N. P., P. Goncalves and L. J. Black (2001). "Past the tipping point: The persistence of firefighting in product development." California Management Review **43**(4): 44-+.
- 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.
- Rifkin, S. (2001). "Why software process innovations are not adopted." Ieee Software **18**(4): 112-+.

- Schmidt, J. B. and R. J. Calantone (1998). "Are really new product development projects harder to shut down?" Journal of Product Innovation Management **15**(2): 111-123.
- Sengupta, K. and T. K. AbdelHamid (1996). "The impact of unreliable information on the management of software projects: A dynamic decision perspective." Ieee Transactions on Systems Man and Cybernetics Part a-Systems and Humans **26**(2): 177-189.
- Sterman, J. (2000). Business Dynamics: systems thinking and modeling for a complex world. Irwin, McGraw-Hill.
- Sterman, J. D. (1985). "A Behavioral-Model of the Economic Long-Wave." Journal of Economic Behavior & Organization **6**(1): 17-53.
- Sterman, J. D. (1994). "Learning in and about complex systems." System Dynamics Review **10**(2-3): 91-330.
- 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.
- Weinberg, G. M. (1994). Quality Software Management: Systems Thinking, Dorset House Publishing Company.