Issues in Strategic Management of Large-Scale Software Product Line Development

by

Jean-Baptiste Nivoit

Ing. Supélec (1997)
M.S. Mathematics, Université Pierre et Marie Curie Paris VI (1997)

Submitted to the System Design and Management Program
in partial fulfillment of the requirements for the degree of

Master of Science in Engineering and Management

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

Author . . . .
System Design and Management Program
May 9, 2013

Certified by . . . .
Stuart Madnick
John Norris Maguire Professor of Information Technologies, MIT Sloan School of Management &
Professor of Engineering Systems, MIT School of Engineering
Thesis Supervisor

Accepted by . . . .
Patrick Hale
Director
System Design and Management Program

Issues in Strategic Management of Large-Scale Software Product Line Development

by

Jean-Baptiste Nivoit

Submitted to the System Design and Management Program
on May 9, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Engineering and Management

Abstract

This thesis reflects on the issues and challenges large software product engineering managers face. Software is hard to engineer on a small scale, but at a larger scale, engineering and management tasks are even more difficult. In the context of software product line evolution, the goal of this work is to look at current managing practice, through the lens of Systems Thinking as well as my own experience. We develop a System Dynamics model to operationalize the notions examined here and run a variety of experiments representative of real situations, from which we learn some lessons and recommend policies that engineering leaders may use to manage large-scale software development organizations.

During the course of this research, we found that the model developed intuitively matched experiences in the software industry. Product line engineering and tighter deadlines force software producers to require more accurate control of the production capability of their development organization. In the context of many release cycles and multiple simultaneously active releases, we present some findings about scheduling of the workload, which the engineering manager may leverage to make decisions about the allocation of work. The research presented here from the point of view of the producers of software can help other stakeholders in the software ecosystem understand the challenges these organizations face and the reasoning behind choices made by these providers.

Thesis Supervisor: Stuart Madnick
Title: John Norris Maguire Professor of Information Technologies, MIT Sloan School of Management & Professor of Engineering Systems, MIT School of Engineering

Acknowledgments

First and foremost, I would like to thank Stuart Madnick, my advisor, for his guidance during this Spring semester. Second, I have to highlight that this work benefited greatly from the help of Allen Moulton, Research Scientist in Professor Madnick's research group, who was a well of knowledge and a very passionate person to discuss issues with. I must express deep gratitude to Allen for his availability and breadth of knowledge, which were more than helpful over the course of this work.

Special thanks also go to other ESD faculty, in particular Dr James Lyneis and Dr Olivier De Weck, who opened my eyes about project dynamics in class ESD.36. I think that, for me, that very class tied the systems thinking view of the world to organizational dynamics, as I was able to recognize phenomena I experienced in my work life. I am sure the teachings of these individuals will be with me for the rest of my career.

I also owe much gratitude to Bill Upham and Derek Coleman, at work, who went out of their way to accomodate for my being away at school and not always available. Going through the System Design and Management program at MIT while still working full time at the same time proved to be a trying challenge, by giving me the flexibility to go to class on work days when necessary, they both helped as best they could to make it possible for me to actually overcome that challenge.

At the time of writing these few words of appreciation, I think of my family, who is far away in geography, but never in thought. Education always was a serious thing in the family, I would like to thank them for that. I would not be where I am today without them. Similarly, many long-standing friends both in France and in the United States have encouraged me to apply to the SDM program, I thank you, you know who you are.

I would like to say a few words about my SDM 2011 cohort, as its members have been a terrific crowd to study with. Quite a few of us that year did the two year commuter option of the program, and two years after the beginning, here we are all at the time of final closure. I enjoyed working with you all, you deserve congratulations for bearing with me during all these months. Special mention to W. Andrew Oswald who proved to be a sounding board in times of uncertainty in my final semester.

Last but not least, I owe most to my cherished Isabelle, as she unconditionally took most (even all, I should say!) of the responsibility at home so I could complete my program at MIT. Merci, ma chérie.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

Chapter 1

Introduction

1.1 Motivation

Software development is hard, and many software projects fail : these are two unfortunate but undeniable truths about the software industry. In the face of these truths, how are managers of software products and product families supposed to steer their development efforts as simultaneously codebases grow and deadlines become ever so tighter? The goal in this thesis is to propose some themes of reflection around these issues, by drawing from the literature as well as my personal experience.

In today's fast-moving world, successful software companies need to grow continuously in revenue, which translate into growth in headcount, market share, product feature set and product line-up. These also form the basis on which software companies compete. Companies that remain small typically merely cater to a niche in the larger market : this is success by one measure, as such niches might indeed be quite large, but this is not the kind of development effort we are interested in in this thesis. Here we are specifically looking at what happens in larger efforts, usually multi-year projects employing hundreds or thousands of engineers. This is the environment the majority of my work experience comes from, and even though managers know instinctively that there are limits to growth, limits to how many features one can cram into a single release of the product, that firefighting is best avoided, that projects easily spiral out of control if the organization promises more than it can reasonably be expected to deliver, all of these things still frequently do happen. This work is an opportunity for me to step back from my daily software development work and reflect on the system issues in the engineering of large-scale software systems, and in particular product line engineering of such such systems.

There are at least two systems involved here:

- the software system produced by the organization, this we call the *product system.* It may be a single system, or a family of related systems.

- the organizational system, that is the organization and its work processes: this is what creates the *products* sold to customers.

This last system is the one we are concerned with most in this work, we call it the *software production system* or the *software delivery system*.

Wheelwright & Clark [Wheelwright & Clark 1994] noted that *being good at product development makes a difference*, and indeed that while having senior managers help on one particular project might rescue a failing effort, what really brings value is when the senior manager can put in place a *system* that makes product development repeatable and reliable. Looking at the levers that make this possible in the particular setting of software product line development is what interests us in this work.

1.2 Research Objectives

Whereas a project has by definition a finite time span, a successful product line has an unbounded lifespan and indeed large software product development organizations typically release their product or product family periodically, hopefully many times if the venture is successful. Each period can be considered a time-boxed project: this insight is the starting point for the research topic at the core of this thesis. Of particular interest are the issues and challenges managers face when the system size increases in number of features, components or people, when the number of parallel projects increases or when release frequency changes, as well as the possible managerial choices and their consequences.

The goal here is not to give a catalog of "best practices" in software development, as there are many such practices and no single one is 'best', by whatever criterion one cares to choose. There exists a whole spectrum of practices: from undisciplined to very disciplined, such as *SEI*'s Capability Maturity Model. The intent here is to describe what the issues are in the management of time-boxed software development, in particular we intend to examine how engineering managers can detect them and how to manage them in an economically viable way.

We shall describe the possible alternative steady states of a large software engineering organization, and try to suggest ways of transitioning from one to the other. Another contribution of this thesis is to look at what long-term effects can result from short-term changes in operating procedures of such organizations. The main point of this thesis is that growing a software product development effort requires stability in change, and that there exists managerial levers that can be used to construct *stable intermediate forms* of the organization, to reuse the words first used by Herbert Simon [Simon 1962]. These intermediate forms allow the organization to achieve growth successfully.

Here is a list of questions we would like to examine in this thesis:

1. What are the advantages of releasing a continuous stream of successive revisions of a software product compared to a big-bang model of product release?

2. How can project managers load-balance between multiple active releases? One example of a product with several versions in active use would be Microsoft®[1]: Windows® Vista, Windows® 7 and Windows® 8 are some of the current releases this company supports actively, shipping bugfixes and updates to customers. This is something that they do today, but their choices are mostly guided by experience and gut feeling.

3. How can project managers load-balance between multiple products in a product family? The canonical example of such a product family would be Microsoft® 's Office® suite: products in that family include the Word® and Excel® software products.

4. How can managers minimize downtime due to human error ? This is similar to "start and stop" fluctations in funding (see [Trammell, Madnick & Moulton 2012]), where the fluctuations are involuntarily caused by the team itself, not by external influences.

5. How to minimize firefighting (as defined in the works of Repenning [Repenning 2001])?

6. What are the cost and benefits of hierarchy and modularization in the continuous delivery process? The point here is that we know the organization typically has or evolves to a structure which is very much like the technical structure of the project, therefore maybe the same principles of hierarchy and modularization can be applied to how work is organized, so as to give the same benefits. One corollary of this is that as one does not want *architecture spanning cycles* [Sturtevant 2013] in the product architecture, one probably does not want such cycles in team dependencies, as that would be a sure way to cause rework cycles that span the whole organization. This also relates to the "*synch and stabilize*" style of development at Microsoft documented by Cusumano [Cusumano & Selby 1997].

7. Does distributed development, especially with regards to remote sites, make large-scale development work more difficult? Under what conditions? How to mitigate the adverse effects of distributing work across locations?

These questions can be examined from the standpoint of a common framework which has its roots in Systems Thinking. In short, we believe it would be useful to construct a model where we introduce some probability of human error,then show that introducing hierarchy and modularity can help control the dynamics engendered by the rework loops that such

[1]Windows®, Office®, Word® and Excel® are registered trademarks of Microsoft® Corporation in the United States and other countries.

errors cause. In other words, dividing the work in modules, sometimes called *work packages*, can help ensure stability in the change process (where "change" is really the "ongoing changes in source code as development proceeds"): each delivery keeps the system under development in a "ready-to-ship" mode, as any intermediate form is a stable intermediate form.

1.3 Thesis Structure

This thesis work aims at providing a good understanding of the dynamics of large-scale software development, in particular how that compares to small-scale development, and what specific issues managers will encounter when growing an organization to tackle ever larger projects.

Chapter 2 will briefly look at some bests practices for software development. This is not meant to be a catalog of all possible practices, but rather an introductory glance at some practices have experienced industry and how they impact software development. The elements presented in this chapter will be used in later chapters to explain some of the phenomena observed.

Chapter 3 attempts to frame the differences between single-person software development and larger efforts. The typical workflow of single-person software development endeavours is presented, which is useful because it really is the minimal system worth looking at, and indeed it is the sub-system embedded in team projects. We then show how each of the practices in single-person development break down as the number of participants increases.

Chapter 4 examines the challenges and issues encountered by large software engineering organizations, such as distributed development, acquisition and merger of other organizations, and integration of product line engineering into the organizational system. These present unique risks and opportunities that leaders must attend to, and we shall demonstrate those in the later modeling chapter 5.

Chapter 5 uses the System Dynamics methodology to present some models which account for the dynamics seen in prior chapters. The goal here is to use SD to gain insight into the system-level issues found in large-scale engineering and to identify the levers managers that can use to keep control of their product development effort. The goal is not to develop predictive tools, but rather to gain understanding of the dynamics.

Finally, **Chapter 6** identifies some interesting topics of future study, before summarizing our findings and concluding in **Chapter 7**.

Chapter 2

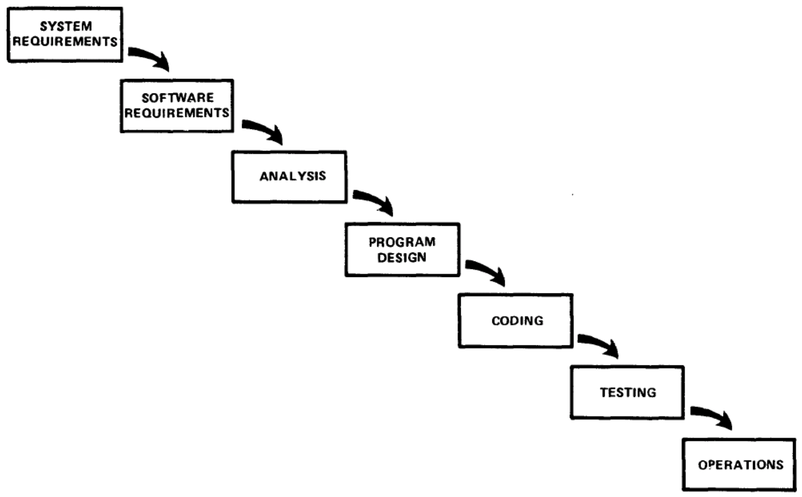
Best Practices for Software Development

In this section, we will look at some of the practices that both individual software developers and team projects use. Many of these are not exclusive to software development, but have roots, or have been independently developed, in other engineering disciplines.

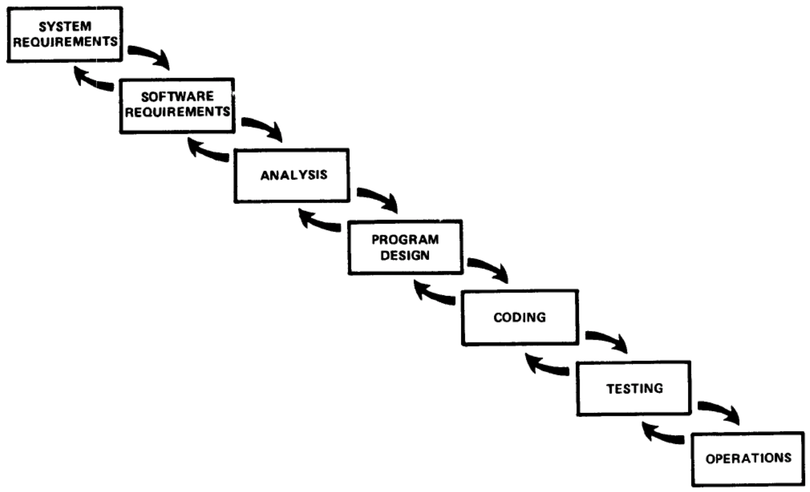
Early work such as [Abdel-Hamid & Madnick 1991] has looked at the dynamics of the software production system, in fact Abdel-Hamid and Madnick found *investigating the policies, decisions and actions that can cause cost and budget overruns in spite of stable user requirements* to be interesting, and that is in fact what we want to look at. We consider all exogenous factors to be stable, and observe that even then the dynamics of software development are such that cost and schedule overruns still do happen. Before attempting to integrate the various factor which intervene in software development projects, we want to look at various aspects of the software production system, viewed as a socio-technical system of growing complexity.

2.1 The Waterfall and other software process models

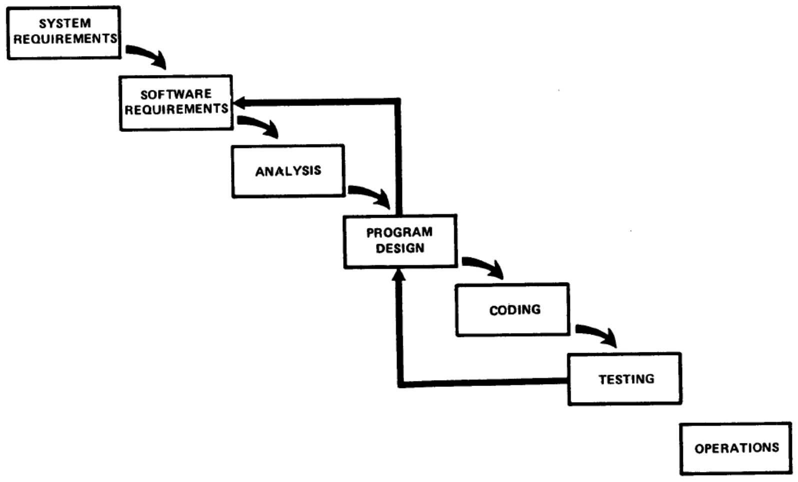
The academic literature presents a wealth of information about the theory and practice of software product development; large parts of this literature is interested in modeling the processes by which an organization produces software products and families of products, but the very roots are in the theory and practice of project management. Indeed developing a product from scratch is often considered a one-time project. One of the first software process models is known as the *Waterfall model*, by Royce, depicted in Figure 2-1a. Winston Royce [Royce 1987] did not coin the name *Waterfall model*, but he did introduce this abstract model of software development in the 1980s.



(a) Implementation steps to develop a large computer program for delivery to a customer.



(b) Hopefully, the iterative interaction between the various phases is confined to successive steps



(c) Unfortunately, for the process illustrated, the design iterations are never confined to the successive steps

Figure 2-1: the Waterfall model, from [Royce 1987]

Figure 2-1c shows that Royce had clearly identified that there was an *iterative interaction* between steps, which we would qualify as feedback. He had even warned about interaction between non-consecutive steps. This is in fact the beginnings of the idea of the rework cycle, detailed in chapter 5.

Early scholars of software project management quickly saw the shortcomings of the pure Waterfall model, Boehm [Boehm 1986] formalized what happens in software projects in what became known as the Spiral model (See figure 2-2). In this early research, we can see that the concept of iterations is crucial to managing projects: in particular, planned iterations are good to have, as having them merely recognizes that work is iterative, but unplanned iterations are the cause of many issues that prevent successful completion. The very name *unplanned iteration* indicates that this phenomenon is unforeseen rework, and as we will see in later chapters, such rework can be controlled. In particular, when it is not controlled and spans many steps of the model, it causes the very schedule problems that plague software development projects.

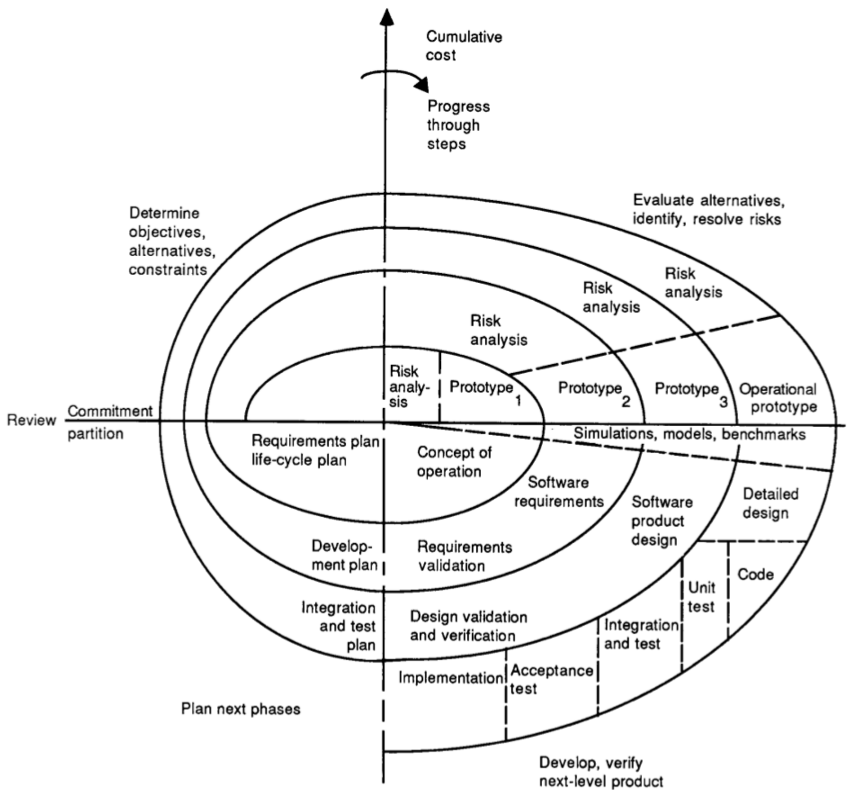


Figure 2-2: Boehm's spiral model, from [Boehm 1986].

These academic process models represent what happens in the development of a software product at a very high-level, here we are not concerned with how close these models the reality of software development, we simply use the theoretical description as a framework for how software is developed in industry. Specifically, in the next sections of this thesis, we will be concerned with what happens in the detailed designed, coding, testing and implementation stages.

2.2 Source Control Usage

Source control is a practice derived from software configuration management. Configuration management is a discipline that records all existing configurations of a (hardware or software, or a combination of the two) system, for the purpose of being able to reproduce these configuration on demand. In the software development industry, configuration management is specifically used to perform the bookkeeping to record all existing versions of a software product.

Each ongoing line of development is called a *release branch* or simply a *branch*. Alternatives names for the same concept are a *mainline* or a *baseline*, all are used interchangeably in the present document. On a given branch, one can make several successives changes, called *submissions* or *promotions*, and the branch represents a timeline of these changes. One can then select to view the state of the files at any given point in time: this gives the ability to go back into a previous state.

Some such versions are "tagged" or "released" to customers: these are the actual artefacts that are shipped to customers, perhaps in the form of bits on a DVD or available for download from a website. Source control is usually implemented using source control software, sometimes called *Version Control System*, such as CVS, Subversion, or ClearCase. It provides to the individual developer an infinite undo capability, and to the enterprise a means to record any and all changes. It will become apparent later in this paper that automated bookkeeping is a crucial capability to have to work on many code mainlines simultaneously. [Berczuk 2002] gives details.

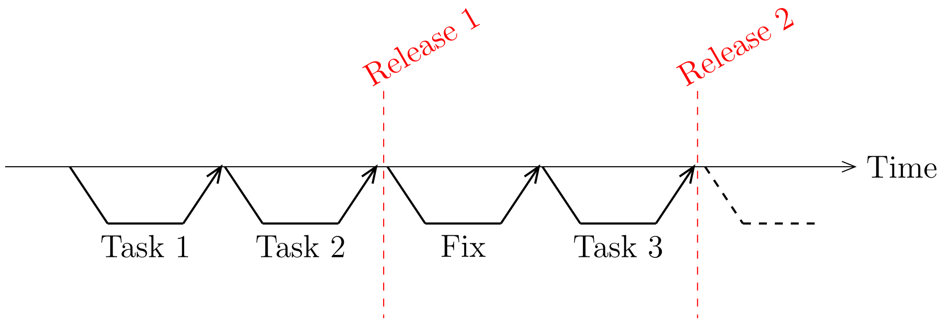


Figure 2-3: This shows how source control is used to make successive modifications to source code, create releases and record the history of changes.

As shown on Figure 2-3, For each coding task (either in initial implementation or in defect fixing), the software developers "checks out" a copy of the source code (i.e. he gets a copy from the source control system, this is sometimes called *attaching* or *creating a view*). He then makes changes to some of the source code files, recompiles, runs the software product to validate his change(s), then runs unit tests, and if good, *promotes* (or *submits* or *commits*, the terminology varies for each SCM system) to make his change part of the mainline. At the time of the final build whose bits will be shipped to customers, a member of release engineering generates a build and that becomes the official build given to customers. Release engineer can be the developer himself, but on typical large teams, there is dedicated staff for this function.

In the context of software product line engineering, it is important that the source control system track changes already merged on each mainline. Managers want to provide an environment where the risk of providing incomplete code changes or incorrect merging of code changes are limited. Some of these advantages can be obtained through tool support, for instance SCM software such as Subversion provide *merge tracking* so the system remembers merges that have already happened, which is very useful when merging repeatedly onto one particular mainline. The source control system also usually has some way to group changes to multiple files and to trace back one submission to a group of files: this is critical in finding groups of related changes that need to be merged atomically onto another mainline.

In my experience, working without source control is like being an acrobat without a safety net, therefore even for personal use, one must have the discipline to use source control and to commit small chunks of work. This practice makes it possible to undo mistakes. Similarly using source control to annotate changes with associated defect informations allows one to trace defects back to specific changes. This makes it possible to go back in time and understand the reasons for a particular change, which helps with future maintenance.

2.3 Defect Tracking

Recording a log of all defects and issues found in the product, whether released to customers or not, is of paramount importance for a software development project. Without such bookkeeping, issues get forgotten, subsequent maintenance finds source code in such a state that the maintainer has a hard time figuring out why the design or the implementation ended up in this state. When a software engineer in a maintenance capacity makes a code change in response to a defect report, he needs to record information about what he understood about the fault, how he gained this understanding, what further assumptions he made: this historical log of everything that concerns a defect may eventually become useful for the next person working in the same area of the software source code. Experience shows that any source file modified for the purposes of repair has a greater chance to see more changes in the future. When this additional work happens, it is called *rework*. *Rework* happens frequently, and therefore it is a wise investment to carefully log changes and their purpose, so that the next maintainer, which may very well be the same person a few weeks out in the future, can put things in context again rapidly.

In short, having proper defect tracking provides the context that is not recorded in the software files and artefacts: this is what maintenance programmers use as inputs to perform their software change tasks.

In a way, the defect tracking system is the repository of the organization's memory of all the defects and associated rework that were ever discovered and performed in successive revisions of the software product. It functions as an engineer's log book, as is used in many other engineering disciplines. It is meant to be a centrally maintained repository of organizational learning.

In the context of software product line evolution, when one organization maintains several mainlines in parallel, defects need to be fixed on each of the active mainlines. As a way to load-balance the work, managers can select one engineer to do the original corrective change on one mainline, and have some other engineer merge the same set of changes on a different mainline. Knowing what files and what changes belong to a particular change set associated to a specific defect is crucial to limit the scope of what the maintenance engineer needs to learn in order to work effectively on the merge. It often happens that a set of changes applies directly onto the other mainline, but it may also sometimes happen that it is not the case: in such a situation, the second engineer needs to devise a different, but semantically equivalent, set of changes: knowing what the original changes are helps frame the problem and its solution, even if the solution needs to be different.

Automated defect tracking also provides for the *Quality Assurance* function, subsequently abbreviated to *QA*, a way to feed their stock of input tasks. QA is the test engineering function within the enterprise: when a change reaches a build that the QA engineers can test, This minimizes downtime for QA engineers (they do not have to ask for status to developers, they do not need to figure out where in the delivery chain the change is), and decreases per-defect overhead for the whole organization.

2.4 Building and releasing software artefacts

During the *coding* phase, individual software engineers contribute source code to the product, but this is usually not in itself sufficient to produce a working product. Organizations typically designate a *build master* to produce the official product build that will eventually be shipped to customers. This operation is usually performed by invoking a compiler that turns software source code into executable files. This technical operation is called *compilation* or *build* by engineers, it may have to be performed once per supported platform (for C/C++ systems) or just once (for Java-based systems). When the product or product family because large enough, the build operation increases in duration, and complete system builds by each engineer becomes impractical. Ensuring linear scalability can be an engineering challenge in itself: if care is not applied, it is easy to create large systems whose build times are superlinear in the number of individual elements. Even with a properly scalable build procedure, whole-system builds require more time than convenient for each engineer to build from scratch every day. In such a situation, the organization will typically have more than one *build master*, maybe a whole team of individual engineers: such a team is called a *build lab* or a *release engineering* team. Another concern is separation of responsibilities: having a separate build team ensures that the work product of individual development engineers is in fact amenable to complete system builds.

What software engineers call *Breaking the build* is the act of rendering the system build procedure unable to operate, this may happen for a number of reasons, such as invalid syntax in sources files, incompatible changes, or omissions of new files.

When one development engineer publishes a change that *breaks* the build, having a second party run the build operation and verify buildability puts in place social processes that incentivizes engineers to fix their own mistakes. The same social processes induce engineers to exercise care in publishing changes, which improves the stability of the build process. Indeed [Dikel, Kane & Wilson 2001, p. 92] make the point that *If the build breaks, the software is in an unknown state, and so larger problems may remain hidden*, therefore ensuring these situations occur as rarely as possible is extremely desirable. The absence of build breaks does not guarantee that the system will work at all, but their presence absolutely guarantees that other problems are present but cannot be discovered in the system. One does not want engineers to work blindfolded, therefore build breaks cannot be tolerated.

McConnell argued in 1996 [McConnell 1996] that using daily builds and daily smoke tests were a good (See next section and also [Sullivan 2001]) way to prevent downstream rework from preventing others on the team from working. Complete coverage is not the goal, a set of simplified tests for the core functions ensures that each build has a minimum level of quality that lets all work continuously.

We should note also that individual engineers usually work with *debug* builds (as opposed to *release* builds, which use different compiler options to produce optimized code), which produce artefacts that can be used with source-level debuggers. Release builds and debug builds are not identical, and sometimes bugs only manifest themselves in one of the two: this happens from time to time and is related to where the compiler places automatic variables, how memory layout is padded, etc... these all make for bizarre memory-related problems which are very hard to debug. Debug builds are fundamentally useful for engineers, while release builds are what customers will really use in production, therefore it is paramount that some, if not the majority, of the testing happen with release builds, in order to be as close as possible to realistic customer usage of the product. When product size increases, it is important to have the ability to mix artefacts from both kinds of build, so engineers can use the mostly-optimized product with only some components in *debug* mode.

In conclusion, when product size increases, building and releasing software products becomes a discipline that requires dedicated staff and policies: Figure 2-4 gives examples of policies companies may put in place to manage this.

In one environment for a C++ product I worked on, in an attempt to limit system-wide breakage, the company had instituted a policy of limiting rebuild impacts on older fielded releases by not allowing changes in *header files* in these old mainlines.

Header files are an artefact of the C/C++ languages, where one file, called the *header*, contains the declaration for a module, and another file contains the actual implementation: when one changes the *header*, callers must be re-compiled to match the possibly changed definition of functions. This kind of impact analysis is performed by the build system to ensure a correct build is derived from the source code. It allows for incremental rebuilds, so that only the parts that may have changed need to be re-compiled.

Disallowing changes to *header files* thus ensured that the externally visible binary interface of any modified DLL remained the same, and that newly rebuilt DLLs could be dropped in in an existing customer installation. By voluntarily limiting change in this fashion, we minimize deployment costs for the customer. In some ways, this policy was too restrictive, as it outlawed many valid changes, but I would observe that it was for the most part very successful in minimizing rework in that specific setting.

In another large mixed C++/Java product release environment, we would know from the architecture what impacts would a header change in the core components have. In particular, if we knew that one such change would cause a large rebuild that would take more than a day, we would voluntarily delay submission in the source control system of these changes to a Friday, so that the rebuild would take place over the week-end and the build artefacts would be ready for the next workday. Release engineers would get to know the system intimately and would schedule builds accordingly, to minimize stalls. For C++ source code, this was usually the result of badly organized header files, proper re-organization usually cut build-times significantly: the engineering aspects of this for C++ systems are largely documented in [Lakos 1996]. Project team who do not engineer this aspect of their production system usually see build-times worsen and unnecessary rebuilds happen more and more often. As builds happen more frequently, scalability and performance of the build system can easily become a bottleneck that limits the ability to generate a new working version of the product.

Figure 2-4: Example of policies regarding building products.

2.5 Automated Testing

Automated testing is an important part of the work of software developers. This is sometimes called *unit testing* in the sense that it tests unitary pieces of the software, however such testing can apply at any level of the system, and can in fact test some very large units. Unit testing may be automated or not, but of course it is best to automate as many tests as possible, which makes it possible for others, such as the release engineering team or continous integration software, to re-run the suite of tests after each code change.

[Abdel-Hamid & Madnick 1991, p. 71] wrote about the coding phase that "*not included in this activity is unit or module testing, which is commonly considered to be part of the coding process*", and indeed that is how we will consider this portion of the testing process in the present work. We should note however that automated testing is vital in isolating the repercussions of problems in the coding phase: software developers are usually not allowed to publish their work to the rest of the team until all automated tests pass. Individuals usually know this from experience and observe a discipline such that errors like these do not block others; even in single-person development, it is good to have known-good checkpoints where testing is performed to verify no regressions have been introduced into the code base.

In [McConnell 1997], McConnell calls this the *Smoke Test*: running the suite of automated tests blocks is done to detect mistakes as early as possible. A software development discipline that lets a broken system be released to other constituencies (be they other developers, testers, or even worse customers) is really no discipline, as it guarantees that every little mishap will stall all the stakeholders. Running the *Smoke Test* is a simple and easy way to force detectable rework to be performed before consequences and cost are too important. We shall see in later sections how more investment in automated test suites can improve the flow of deliveries through the code production system.

2.6 Manual Testing

Some tests cannot be easily or cheaply automated, and must then be performed manually by QA staff. In particular this includes some User Interface tests, or tests that require some interaction with a physical device (for instance, in CAD software, tests that generate code for an NC machine), or system tests that necessitate the presence of many moving parts. The enterprise should strive to automate everything that can be automated, but that will certainly leave some things which cannot be automated.

Manual testing is an expensive kind of testing, as it requires human labor (and it is repetitive labor, making it usually not very appealing to quality engineers). Experience shows that quality engineers will usually come up with inventive ways to automate their testing. What is left is things that must be truly tested by hand, hopefully those are very few.

Manual testing is nevertheless invaluable, because humans interacting with the system can usually be creative in their interactions, in ways that an automated test harness never could. QA engineers also usually come up with contrived, but theoretically possible, scenarios that exercise the system in unexpected ways, often revealing unintended defects or limitations. One should also not dismiss the possibility of find problems by accident when interacting with the system.

While automated testing can rapidly reveal coding errors by uncovering regressions, manual testing usually reveals defects either in design or in coding. Of course, for that kind of testing to happen, one needs to constantly have a *minimum testable product.* The *minimum testable product* phrase used here is pattern after the *MVP* or *Minimum Viable Product* mentioned in literature, here we use it to describe a testable artefact, regardless of its actual viability as a commercial product. This highlights the importance of building as often as necessary, as was explained in Section 2.4. Having a working build which has passed the automated test suite successfully is a requirement for manual testing: the consequence of not having both of those two requirements fulfilled would be that the testers either do not have a testable system, or are wasting their time testing a build which is known to be defective.

It is extremely important that the person doing the manual testing be another individual than the ones who did the design or the coding: because that third-party does not share any assumptions or prejudices, he or she will usually be able to pinpoint limitations that neither the designer nor the coder thought about.

2.7 Metrics

Ever since software has become an engineering discipline, people have invented and used all sorts of metrics about software source code. Humphreys [Humphreys 1989] formalized why and how to collect data about software, explained how measurements can help understand and evaluate software code bases. Measurements can be used by management:

- for preventing some classes of errors. Static source code checking can help detect some classes of errors, especially non-sensical statements in source code which can only be typographical errors (The canonical example is "`if(a=12) then...`" instead of "`if(a==12) then...`"). Automated detection and classification of errors like these allows the manager to see how many occurrences exist, and to prioritize compared to other tasks.
- for decision-making about the readiness of the software for release. For instance, defect density can be estimated and a threshold for approving release or not selected.
- for budgeting purposes,
- for building confidence in the output of the software production system.

However, the manager ought to be careful about setting up automated measurements of attributes of the software products, for *Management By Objectives* often causes individuals to optimize to the specific measure, which can adversely affect the actual objective. In general, as [Humphreys 1989, p. 330] notes, using such data to evaluate people generally backfires; [Grady & Caswell 1987] makes the same observation, indicating that the data *must be used to evaluate the process, not the people.* For instance, we have seen in industry that if people are asked to review the code and put some markers in the source code files, and if the only thing measured is the presence of those markers, then markers will abound, but little code reviewing will take place. In other words, gauging work by exclusively looking at metrics is a sure way to encounter policy resistance: people will optimize to the metric at the cost of lesser overall system performance; this is an example of local optimization leading to global pessimization.

Although the initial intent is to improve the quality of the work product, using source code metrics to drive the engineering work can also sometimes cause perverse effects. One example is explained in detail in Figure 2-5, another example would be when people optimize to the metric: for instance by putting too many or too few statements on a single line to attain a specific number of lines of code, by adding in dummy comment lines to increase the amount of commenting in the software source code.

As any software code change is a liability, the engineering manager should be be sure to make the organization perform work to satisfy metrics goals (an example of non-customer visible technical feature) at the beginning of a release, rather than at the end. When this policy is not observed, changes that add no value are made late in the release cycle, increasing risk. This is one kind of work that can easily be spread over multiple release cycles so as to minimize risk of de-stabilization. Because such quality-related rework must be spread over time, it is important to maintain a repeatable suite of unit tests and keep records about the reasons for each test case, as well as its behavior over time [McConnell 2004, ch. 25].

[Humphreys 1989, ch. 16] and [Grady & Caswell 1987] explain how to establish a software quality program based on metrics and measures.

One experience I had was when the company had long-term stability problems in our software: it would crash after a few hours. This was often because of de-referencing null pointers in C++ software code, which happens when logic errors in the code lead to the program execution failing.

In order to increase the Mean Time Between Failures, management decided to make engineers add non nullity checks on pointers late in the release cycle. This caused a lot of churn in source code and greatly de-stabilized the product. One of the reasons was that checks were added in the form of conditional statements instead of assertions: this caused the system as a whole to be more tolerant of illogical states. Using assertions would have forced an immediate crash of the product, and thus no immediate improvement of the Mean Time Between Failures, but it would also have forced the individual engineers to get to the bottom of each problem, rather than make those problematic situations acceptable. Blind application of a policy in this case resulted in a system becoming vastly more difficult to understand.

Figure 2-5: Example of policy resistance in the use of software metrics.

2.8 Modularity

Modularity in software systems is both an architecture-level feature and a component-level feature. [Parnas 2001] gave an early demonstation of the applicability and usefulness of the concept of modularity to software, but the best characterization of the desirable characteristics of modules is given by Sturtevant (in [Sturtevant 2013]):

> *Robust modules have the property of 'homeostasis' - their internal functioning is not easily disrupted by fluctuations in the external environment.*

Other product development disciplines have discovered this in the form of the Design Structure Matrix. In software development, modularity manifests itself in the way the software source code is organized.

- in C/C++ projects, the artefacts produced by the build procedure are usually DLL (an acronym for Dynamic Link Libraries) files or executable files. As the system is decomposed into modules, each module ordinarily maps to one such artefact.
- in Java projects, the artefacts produced by the build procedure are usually organized in JAR(Java ARchive) files or WAR(Web ARchive) files. Each JAR file depends on a set of other JAR files to build successfully.

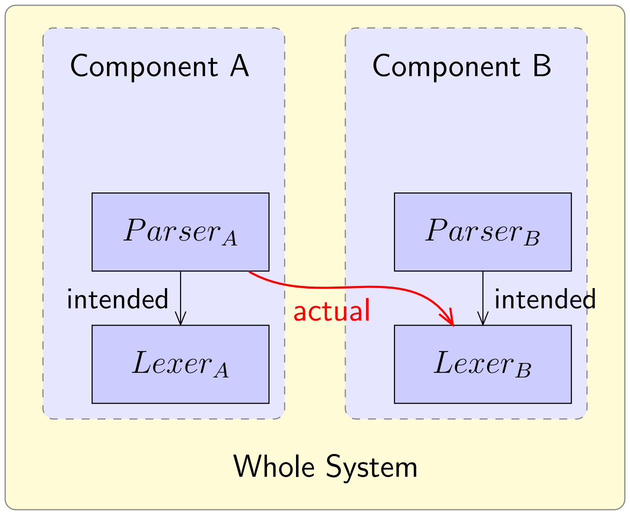
Such an organization then allows individual developers to build parts of the system without having to build the whole product, which is advantageous since it allows to work on parts of the system locally without having to generate a full system build, which may be infeasible time-wise.

Using the Design Structure Matrix, MacCormack has shown [MacCormack 2006] that a modular structure is desirable for large system (although it is not always present), and [Akaikine & MacCormack 2010] showed that it helps bring down maintenance costs over the long term.

One topic related to modularity is that of compatibility at the level of component interfaces: modularity is preserved in software when source compatibility and/or binary compatiblity are preserved. Binary compatibility is the ability to change the implementation details of an interface without changing the interface, such that another component can call into the component being modified without requiring modification of the call site. Source compatibility happens when the interface remains valid at compile-time but not at runtime, this can happen in C++ or Java when a signature changes. Examples of the two are given in figures 2-6 and 2-7.

Modularity is sometimes breached in non-obvious ways, for instance in one C/C++ product I worked on, we had multiple parsers and lexers in different components, those were generated using compiler-compiler tools such as *yacc* and *lex*, which are old Unix tools which generate non-reentrant code, often functions which have a fixed name.

It turned out that on some platforms the visibility of symbols at runtime depends on the order of loading of components. Understanding what code was generated and why this was happening generated a lot of unplanned rework. The solution was to modify the output of the code generator to ensure unicity of names.



More modern languages such as Java provide protection against such accidents, by natively enclosing components in packages. Other C-based system use process isolation to enforce the same constraints and isolate faults so that failure in one component does not bring the whole system down.

Some Java environments such as OSGi go even further and allow multiple versions of the same component to run within the same system: this kind of capability is very important to open systems which are meant to have a long life cycle, to host multiple applications inside the same container system, or to be able to upgrade components while the system is online.

Figure 2-6: Example of binary compatibility issue.

If source code changes are made that break compatibility, then interfaces between components change, possibly without of the client components being updated simultaneously. In other words, if compatibility is not preserved, then updates to interfaces must be made simultaneously with updates to all clients of the interface. This is one kind of architecture-spanning cycle, i.e. a dependency chain that spans more than one component. The best practice is for engineers to avoid making source code changes that break source or binary compability intentionnally.

- removing one function without having first migrated all clients of that function will break the build in the client components (note how this is made even harder when clients can be customer code not available for migration. Typically the organization must then establish some Service-Level Agreement like policy). This is true at the language level, but also true of any interface (either a file format, a network interface, any SOA service, etc...).
- changing one function to have an incompatible signature (in C, this can be source-compatible because C does not check argument types, in C++ or Java, the client component build would break, allowing detection).
- changing one function to have a source-level-compatible but runtime-incompatible variant (for instance in C, changing one parameter type from “int” to “char”, thereby generating runtime trunction of argument values). Another example of this is detailed in the next box.

Some systems enforce compatibility by disallowing the changing of interfaces once published: Microsoft COM is one such system, it has a policy of never allowing the mutation of an interface, enforcing the versioning of the interface instead [Box 1998].

One anecdote I can recount on the topic of binary compatibility as it applies to modularity and coupling is the following: We had a large Java system where we changed the signature of a method in a core component from this version (version 1):

```java

public class Example {

    public static void fct(String fmtstr,
                           String arg1, String arg2) {
        ...
    }

}
```

to that version (version 2):

```java

public class Example {

    public static void fct(String... args) { ... }

}
```

As one can see, this change is source-compatible (i.e. one can rebuild other components that depend on this class without problem, the compiler automatically knows what method to call). For instance, we had callers such as the following:

```java

public class Caller {

    public static void m() {
        Example.fct("%s␣%s","hello", "world");
    }
}
```

While callers compiled correctly with version 2, components compiled with version 1 could not run with a binary core component at version 1 and vice-versa. In this instance, breaking binary compatibility made it impossible to run different versions of the coupled parts together.

Figure 2-7: Example of source compatibility issue.

2.9 Defect Discovery

In this section, we describe how defects are discovered in software systems. A *defect*, sometimes called a *bug*, is a flaw in the software system that causes it to operate incorrectly. End-users of the sofware system see *failures* or *faults*. These failures may be rooted in *defects* or in incorrect operation of the software system. The software development organization receives from its end-users reports of *failures*, which it classifies into *defects* or not. Engineers and quality assurance personnel may also report *failures*. The organization also performs triage according to criticality of the defect: this prioritizes the defects. Once a defect has been assigned to a software engineer, that person will attempt to understand the nature of the failure and to locate in the system where the fault lays. Many times, but not always, the location will be in the source code of the software. Once location and cause are understood, the software developer will engineer a fix, which is a source code change that eliminates the cause of the error. This is one form of *rework*. Once that source code change is published to the end-users, the original reporter of the failure will re-test and acknowledge that the change fixed the erratic behavior.

Note that defects may be implementation defects (i.e. problems in the specific expression of the software source code), or may be design defects (i.e. the software operates according to specification but the specification is incorrect with regards to the actual requirement), or even requirement defects (i.e. the requirement is inaccurate or incomplete). Undiscovered defects can lay dormant in the software for a very long time, for instance in rarely used code paths (maybe one needs a contrived scenario to trigger the failure). Such undiscovered defects are part of *undiscovered rework*, although a fraction of *undiscovered rework* will in fact never be discovered and thus never cause actual rework to occur.

As documented by [Pressman 2000], it is widely known in the industry that finding and correcting defects as early as possible is a good practice to have. [Abdel-Hamid & Madnick 1989, p. 39] highlights that

> *the longer an error goes undetected, the more extensive*
> *the necessary rework and the greater the cost.*

The cost of fixing a defect increases as the issue is discovered later in the project lifecycle: for implementation defects, discovering defects in the Quality Assurance phase is more costly than discovering during initial development; for design or architecture defects, the cost of discovering and fixing after the product has been fielded can be enormous (see Figure 2-8). In System Dynamics parlance, there is a feedback loop, and delays in discovering rework downstream can generate enormous waste in terms of work spent on defective work products and downstream re-work required after upstream corrective action.

Code reviews, also called inspections or walkthroughs, are a well-documented practice

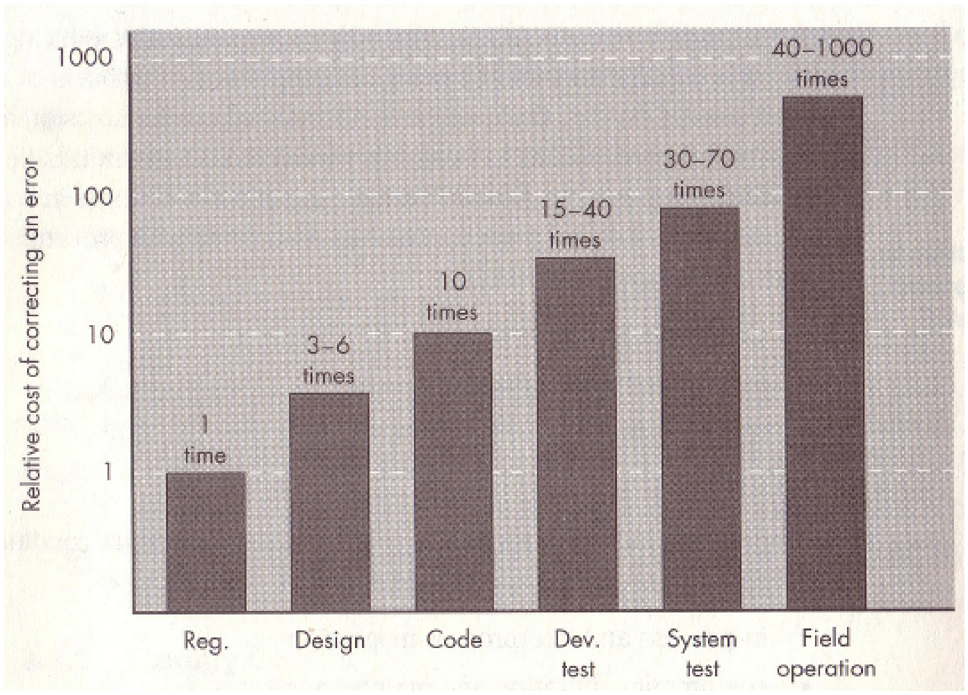


Figure 2-8: Relative cost of correcting an error, from [Pressman 2000, p. 198]

(See [Yourdon 1989] and [Humphreys 1989, ch. 10]) that improves the quality of software by detecting defects early. However code reviews can become counter-productive when their spirit and goals are not assimilated into the organization: for instance, in one organization I know, the mandate to do code reviews was put down one day, and rapidly developers were either stamping their review approval without even reading the source code change, or noting issues but not fixing them (as that would have required another round of work and another round of code review), or even detecting issues that were not present (for instance detecting a misuse of a particular class that was in fact not a misuse at all!). This perversion of the original spirit of the code review process completely made this new process devoid of meaning and devoid of actual utility: worse, it even caused unnecessary and invalid rework. Ideally, code reviews should be conducted for each source code change, no matter how seemingly insignificant; however this would turn out to be prohibitively expensive, if only in terms of the paid time spent by engineers performing the review. In practice, high-level code reviews should be conducted to validate designs before implementation, and engineers use their best judgement to elect to have some changes reviewed and others not, usually as a function of the complexity of the change, their own familiarity with the portion of the codebase being modified. Unfortunately, as [Abdel-Hamid & Madnick 1991, p. 71] notes, inspections are often the first practice to be abandonned when managers start cutting corners.

2.10 People are not always interchangeable

It is well known that individual productivity for software developers can vary by as much as an order of magnitude.

Also, Sturtevant found that engineers working on more complex parts of a software design are less productive than others working in less complex parts [Sturtevant 2013]. This would tend to indicate that engineers working on the core of the system should not be expected to be as productive as others working on peripheral components: in other words, the release pace of the core component will probably be slower than that of the outer components.

As a last note, we should also remark that people are not even interchangeable with themselves: the productivity of an individual is not constant over time, it tends to vary with project experience, schedule pressure, fatigue, morale and motivation.

Chapter 3

Scaling from one-man development to many-engineers development

Scaling can happen on many dimensions, including code-size, headcount, number of products. Scaling in size of code usually happen simultaneously with an increase in headcount, as more people are required to develop and maintain a growing code base, and conversely one would expect more engineers produce more software code. The number of products is correlated with the code size, but not necessarily directly proportional : many products may be simply rearrangements or bundles of features, or different skins on the same package, as can be the case when several industry vertical products can be derived from a single product simply by adopting the specific language of that vertical.

Single-person software development is typically nimbler than large projects, as the cost of mistakes is usually smaller. Although such endeavors can be disciplined, they typically take many shortcuts, which may very well be acceptable since the expected deliverables really are not the same. In a one-man project, developer, manager and strategist are all but the same person; in bigger projects, these roles are distributed among many: this requires coordination.

As Weinberg points out in [Weinberg 1991, p. 67], what is desired here is an *engineering discipline*, not *hacker* or *hero*-like behavior. If we look at Weinberg's classification of thought patterns in software management:

These thoughts patterns parallel the levels of maturity defined by the Capability Maturity Model [Paulk, Curtis, Chrissis & Weber 1993]. Single-person projects usually are at Pattern 0, in this terminology.

Pattern	**Name**	**Description**
0	Oblivious	Individuals do whatever is necessary to get the job done.
1	Variable	Individuals do what they think is necessary at the time to get the job done.
2	Routine	The organization follows a routine, except in times of difficulty.
3	Steering	The organization has selected a process for its good results, and follows it.
4	Anticipating	Processes are established based on past experience
5	Congruent	Processes are repeatable, but constantly monitored and improved.

Figure 3-1: Weinberg's classification, from [Weinberg 1991]

Delivering a single-release product for a very specific purpose is very different than delivering a large product or a product family. The software code production system consists in the whole chain from requirements collection to the final mastering of the actual bits of the software products. It is a complex socio-technical system, one that may be viewed through the Systems Thinking lens. Single-person development efforts can certainly be looked at from this perspective as well, but the more complex the system becomes the more important the holistic analysis becomes.

Some of the problems that software development organizations encounter when increasing project size are:

- increasing headcount induces a greater need for coordination. This coordination may be related to attributes of the product being developed by the organization, but it may also very well be related to communication to all involved parties of what the accepted work processes are. There is tremendous value for the organization in documented and standardized work processes. As [Abdel-Hamid & Madnick 1991, p. 51] points out: *Brooks suggests that human communication in a software development project is the most significant cause of overhead - i.e. slowdowns and obstacles.* Another point is that increasing headcount makes the situation described in Figure 3-9 much more frequent: instead of one person doing two separate tasks and knowing the impact of one on the other, now every pair of developers must know about the possible interactions of changes made by the individuals.

- increasing headcount can have the counter-intuitive effect of decreasing the work output of the organization in the short term. This is called *Brooks' Law* and is described in section 3.1.1.

- increasing the code-size or the number of components causes build times to make complete system builds too costly and infeasible for individual engineers. This is described in section 3.3.3.

- increasing headcount makes occurrences of human errors more frequent. One engineer's change can prevent others from working, and require emergency fixing. In other words, more workers cause more stalls in the software code production system.

- increasing the number of components increases the number of interfaces, and more interfaces means more opportunities to break modularity, which in turn can increase stalls.

- increasing the number of interfaces means that formal protocols must be in place to evolve these interfaces without breaking the work product. Another way to put this would be to say that whereas in a single-developer system, only the customer cares about compatibility at the time the product is fielded, in many-developers efforts, one must continuously ensure stability in the form of compatibility.

- increasing the number of simultaneously active mainlines, in other words the number of branches which are kept supported and maintained by the organization for customers using those in production, yields more opportunities for rework to propagate across mainlines.

3.1 Human Resource Management

The first aspect that we want to examine is that of management of human resources in the context of a large organization. This topic has been looked at extensively in the literature, in particular [Abdel-Hamid & Madnick 1991] calls it the *Human Resource Management sector*.

3.1.1 Brooks' Law

Brooks' Law [Brooks 1975] states that *Adding manpower to a late software project makes it later*, and indeed this is an occurrence of the *Limits to Growth* pattern documented by [Novak & Levine 2010]. The organization viewed as a system can only accomodate for integrating so many new people in the software development project, as adding more causes old-timers to spend more time hand-holding newcomers and fixing defects caused by the same than producing deliverables.

3.1.2 Hiring

Apart from the problem of adding staff in hopes of getting work done, the organization needs to deal with the natural attrition of staff: as people leave the company or retire, new hires must be brought in to ensure the development capacity is continuously replenished. In the case of an organization producing a long-lived product family, the time horizon for hiring can be long, possibly measured in years. For all intents and purposes in this paper, given that the life cycle of software products examined here spans many years, we shall consider that the number of people on staff is maintained constant, which is an approximation of reality but should suffice.

Nevertheless we should note that longer tenure also engenders natural attrition of engineering staff, and therefore management must take this into account and make sure to replenish positions as needed, while accounting for the time people need to acquire experience with the specifics of the product family and the design of the architecture and components.

[Dikel, Kane & Wilson 2001, p. 92] shows that the release rhythm of the software development organization can have impact on other functions. For instance, the Human Resources department may need to coordinate activities such that interviews happen at a time when the development teams are not overloaded with work (e.g. not at the end of a release cycle), and such that hiring and bringing new staff in happens at a time when the team can incorporate new members.

In the present modeling effort, we do not consider this point and instead just assume that hiring to replace attrition happens smoothly over release cycle. Of course hiring at a bad time could cause disruption to the software delivery system, we just do not consider those effects here.

3.1.3 People

One damning sign of one-person or few-persons organizations is that they tend to rely on heroes to perform extraordinary work to save the day on the engineering side. However, as teams grow, the company cannot rely on heroes anymore ([Bosch 2000, p. 316]). To do so would be encouraging and rewarding fire-figthing. We can observe that the firefighter, sometimes called a *software hero*, is a critical resource in the critical chain method of project management. When the organization uses the critical path method of project management, if one resource (i.e. one firefighting programmer) is made unavailable (i.e. assigned temporarily to some other project), then the manager unwillingly has transformed his project into a critical chain.

This is not to say that one should not recognize that abilities vary from one individual to the next, merely to say that the growing organization must channel the energies of its top people in a way that is not counter-productive. Relying on heroes and firefighting is one of those addiction phenomena Weinberg cites in [Weinberg 1991, p. 154].

3.1.4 Organization

With regards to organizing work, many software development entities attempt to use various patterns of organization, including those documented in the *Organizational Behavior* literature. Each way of structuring people has its advantages and its drawbacks.

[Pohl, Böckle, & van der Linden 2010] have summarized the various generic forms of organizations, as they apply to human groups working on the engineering of software product lines (see Figure 3-4).

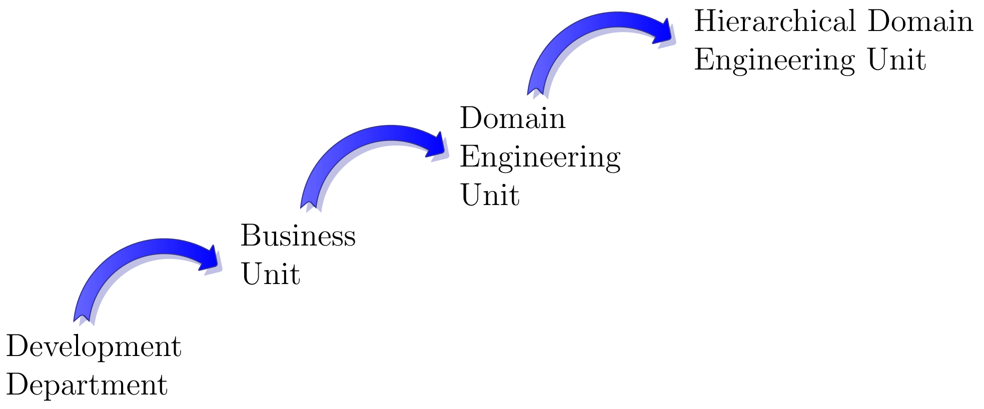


Figure 3-2: Successive organizational forms as the system grows, after [Bosch 2000]

[Bosch 2000, ch. 14] describes a clear progression in the possible form of organization as product line size and complexity increases:

- *Development Department* is the first form, corresponding to a single, possibly quite large, group devoted to development and engineering,
- *Business Unit* is when the development sub-organization becomes as important as other functions within the company,
- *Domain Engineering Units* are found when some "core" assets (part of the *domain*) are developed in reusable fashion, with the explicit goal of providing a shared platform for application engineering,
- *Hierarchical Domain Engineering Unit* is when the engineering unit is itself organized hierarchically in sub-systems. Hierarchy is used to distribute labor and realize the layered dependency structure of a product line divided into domain and application components. For very large products, this is the only workable form of organization, and is shown on Figure 3-3.

These forms are shown in Figure 3-2.

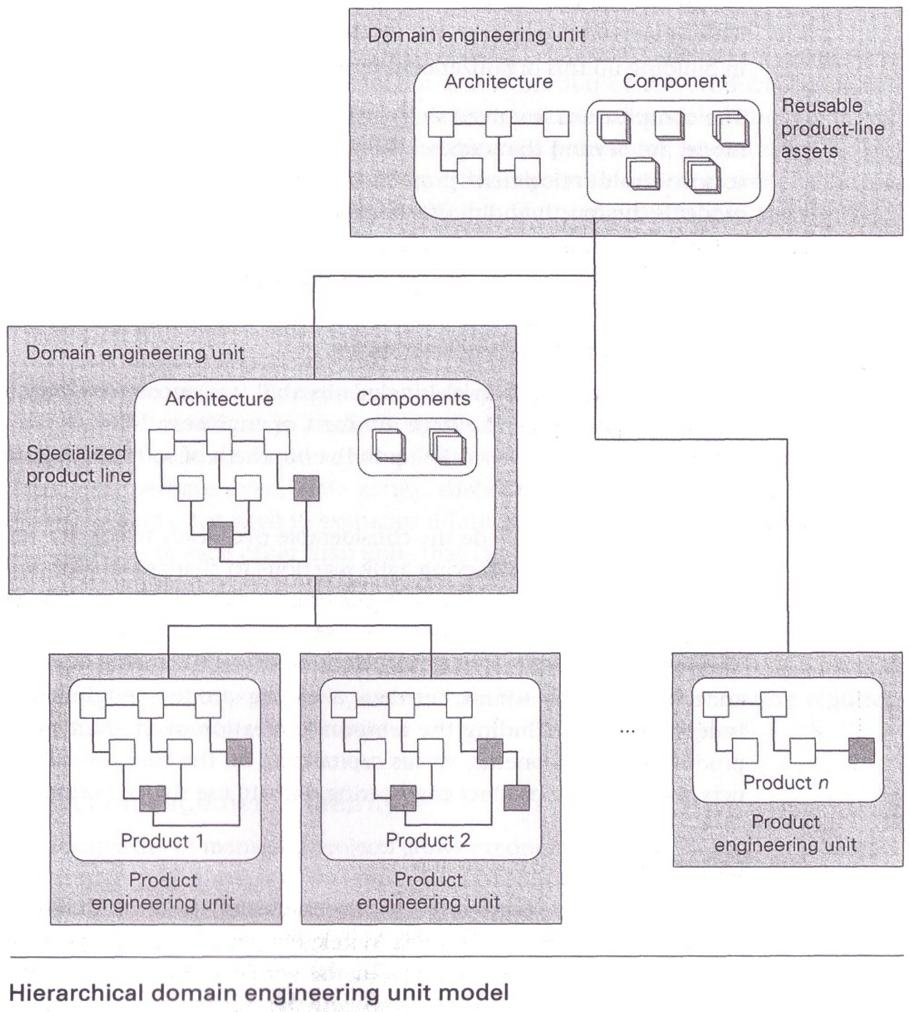


Figure 3-3: Hierarchical domain engineering unit model, from [Bosch 2000, p. 313]

Using a different viewpoint, [Pohl, Böckle, & van der Linden 2010, ch. 19] describe the canonical *matrix* organization of labor used when some staff is used to do technology development, but loaned to projects on the short term. The matrix organization in all its variants exhibits the well-known tension between the immediate need for solution in product development, and the long-term learning needs of technology development. These are shown in Figure 3-4.

When applied to hierarchy in software development, this already indicates that the speed of advancement of the shared parts of the product is slower than that of the application-specific parts of the product. [Pohl, Böckle, & van der Linden 2010] does not explicitly describe the hierarchical domain engineering unit model from [Bosch 2000].

These possible forms of organization are described in terms of the structure of the product line, but the literature usually omits the time dimension, that is to say it does not formalize

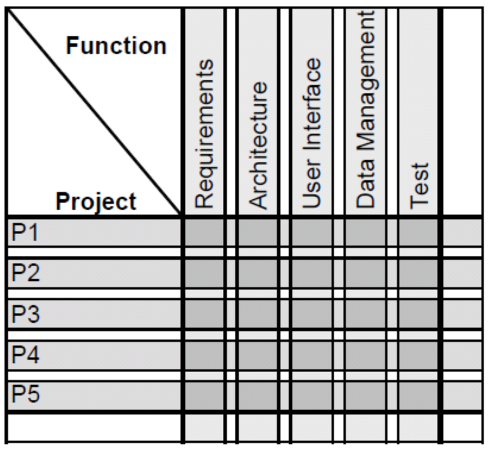


Fig. 19-4: A matrix organisation

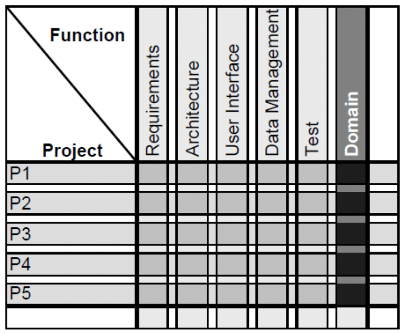


Fig. 19-5: Matrix organisation with domain engineering as functional unit

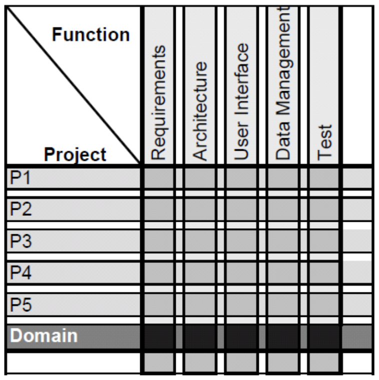


Fig. 19-6: Matrix organisation with domain engineering as project unit

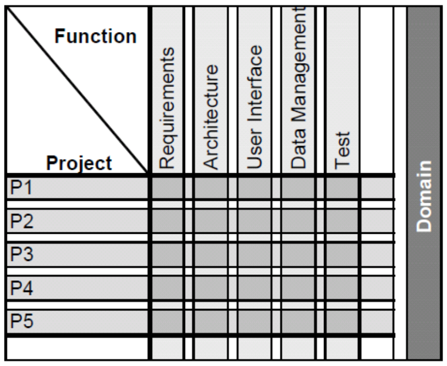


Fig. 19-7: Matrix organisation with separate domain engineering unit

Figure 3-4: Possible matrix organizations for product line engineering, from [Pohl, Böckle, & van der Linden 2010, ch. 19]

how to organize work in the context of product line evolution, how to manage the various stages of the lifecycle of both products within the product line and product line itself. [Bosch 2000, p. 308] does mention the risk of *erosion of the architecture and components in the product line* if ownership of components is not managed carefully (see Figure 3-5):

> *The timely and reliable evolution of the shared assets relies on the organizational culture and the commitment and responsibility felt by the individuals working with the assets.*

Here we should make a note that the academic literature has no canonical terminology for the various pieces of the product system. Indeed MacCormack, Pohl and Bosch all use different terms to designate roughly the same things:

– what [MacCormack 2006] calls "core", [Bosch 2000] calls "shared assets" and [Pohl, Böckle, & van der Linden 2010] call "domain". The "core" (or "domain", sometimes also called "platform") is the central component applications are built on top of, this is what is reused across products.

– what [MacCormack 2006] calls "periphery", is called "application" by [Pohl, Böckle, & van der Linden 2010]. "Periphery" or "application" are the user-actionable part of the software.

Both [Dikel, Kane & Wilson 2001] and [Allen 1984] also documented the often forgotten informal aspects of how work gets done within software development organizations: indeed, while many organizations choose a hierarchical structural decomposition, knowledge is often transferred in ways that do not conform to the prescribed boundaries and channels of a rigid hierarchy. Always using formal channels slows communications down, creates coordination where none is required, and generally slows down the pace of work, as shown in the recent revival of *agile* methods, which mainly argue for nimbler and simpler processes which spend time and resources only when necessary. As noted in Section 2.4, another informal aspect of people interactions is the social pressure some environments associate with the blame for *breaking the build*, a soft way to ingrain the practice of caution when submitting changes to software source code.

[Bosch 2000, p. 308] notes that the best organization for a given product line depends on its size, its genesis and its goals. Breaking down into organizational units may be required to keep coordination needs reasonable: units too small can make it necessary to have as many communications as the square of the number of workers, which clearly cannot scale. At the other extreme, very large organizations need to locate ownership for the core components of their product separately from ownership and responsibility of product development, otherwise the integrity of the architecture may be violated, causing the core components to depreciate in value over time.

The *Hierarchical Domain Engineering Unit*, as shown in Figure 3-3, usually is a sign of maturity of both the organization as it moved to hierarchical decomposition as a way to control and promote reuse, and a sign of maturity of the software assets themselves, as successful reuse requires significant effort. This hierarchical decomposition can have arbitrarily many levels, each of which is associated to a set of assets: the leaves of this tree contain the non-shared parts which are product-specific, while the intermediary nodes within the tree represent some level of sharing of assets.

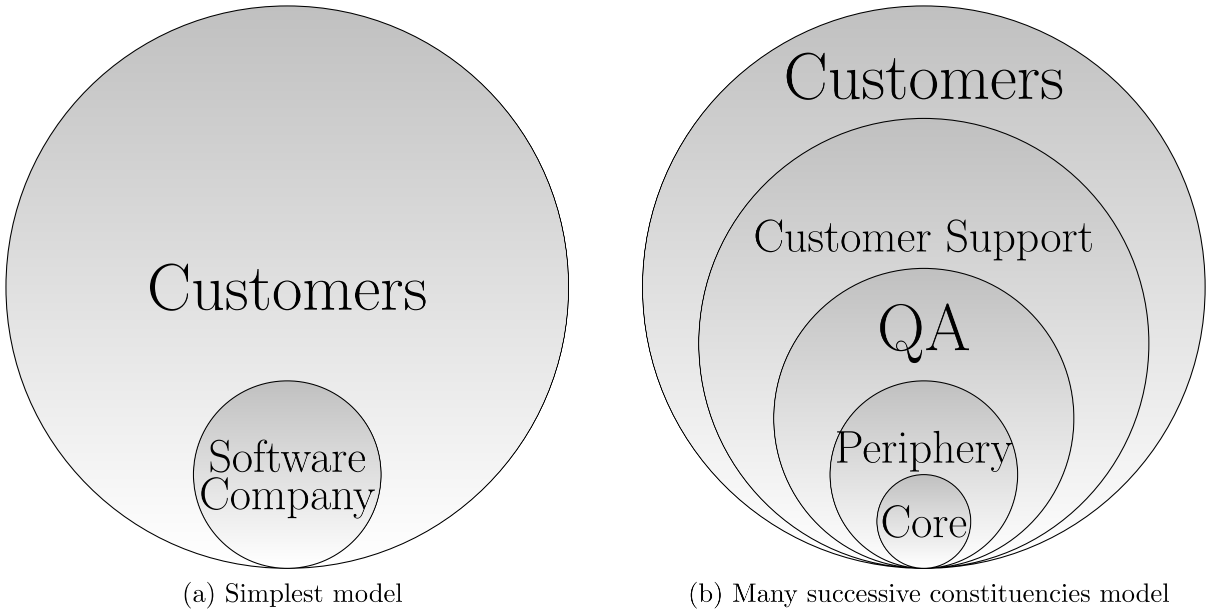
The organizational structure and the asset structure both evolve, and they do independently. Experience shows that this causes issues with actual ownership of assets. Ownership is more sticky than one would like, as knowledge and experience with both the design and the implementation of the source code component tends to reside in engineer's minds as mental models built over time. For this reason, ownership of components stay with the original developer until someone willingly accepts the burden of maintenance of said components. Transfers are difficult, they induce an additional unbudgeted burden for the new owner. As some markets or applications become more attractive, people transfer to other teams to follow the strategic interests of the company. Components that fall out of favor (maybe because of changes in technology, for instance), tend to not attract new owners. This presents a dilemma for the company, as sometimes stable assets are owned by teams which are difficult to staff. The interests of the company is in stabilizing assets and maximizing the corresponding return on investment, which is at odds with the human tendency to want to work on the "latest and greatest" piece of software. Such issues are largely dependent on people and company culture, therefore it seems that there is no single solution that will fit every company.

Figure 3-5: Issues with code ownership.

Regardless of the actual organization selected, we should note that as the product line becomes more successful, changes become more difficult to incorporate in later revisions of the product line, as care must be exercised to preserve existing function and value. The way this manifests itself is that as the product line and the company grow, more intermediary functions are set up between the product developers and the end customers. Indeed in a small company, the developers might be directly in touch with customers, they might be able to deliver features or bug fixes to the field very rapidly. This approach does not scale, as individuals usually cannot exercise the level of care required to field stable products to customers. In a larger organization, many layers exist and must be crossed for the work product of engineering to be made available to customers. As an example, only a few stakeholders are depicted on Figure 3-6, that set could include:

- the Core development team
- the associated Quality Assurance team
- the Periphery development team (for simplicity, we assume here that there is only one, there could be many).
- the associated Quality Assurance team
- the system Quality Assurance team, which performs complete system testing.
- the customer, here assumed to all want the same work product, but in a real product line setting, there would be many products, and many associated sets of customers (each with their own expectations of feature content and schedule).

In effect, starting with the engineers, each successive function fields its work product to an ever-larger set of interested stakeholders, ending with the customer. This is required to ensure quality of the work (this is for things flowing from the inside out), and to ensure isolation from frivolous or duplicated requests (this is for inquiries flowing from the outside in) and to protect the integrity of the product line vis-á-vis the vision selected by the company.



(a) Simplest model (b) Many successive constituencies model

Figure 3-6: Possible organizations

One important consequence of this is that creating a new version of the product line (i.e. generating a new released mainline) is subject to multiple hand-offs between each of the layers illustrated in Figure 3-6. The successive layers displaed on Figure 3-6a are merely an example, as some organization may elect to include their salesforce as one stakeholder, or add more layers, for instance by having multiple levels of support: *Level 1* to perform fault triage and *Level 2* to get access to more technically savvy personnel.

Each hand-off presents a danger if rework is discovered after the hand-off: when this happens, the work product is in a sense returned to its originating place in the organization and must be analyzed, possibly modified, re-tested and re-delivered once again. When defects cross the boundary between two layers, then the organization has created an opportunity for a rework cycle with a built-in delay to happen, and as we shall see in chapter 5, these are the events that can expand the product delivery schedule very significantly.

3.2 Standardization of processes

As the number of people involved in the development of a product family increases, the enterprise must standardize processes. [Humphreys 1989, p. 393] noted this in the general context of large software systems. This is required to minimize setup time for engineers and for testers, it is necessary to build confidence that the work product of those knowledge workers is a quality product and will function as specified on the target customer system. One key to minimizing the cost of rework is to provide for an environment that makes problems reproductible on any system: the most difficult bug to correct is the one that the engineers cannot reproduce, this is sometimes called an *heisenbug*, defined as a bug that disappears when observed [Gray 1985]. For large classes of faults, reproductibility of failure requires standardization of the system, not only the product system as used by the customer, but also the software delivery system as used by engineers to create the product system. Note that standardization does not *guarantee* that bugs will be reproducible, in particular timing-dependent faults can be tremendously difficult to reproduce, but absence of standardization does guarantee that reproduction will be difficult.

The organization can reap great benefits from having standardized defect reporting procedures, standardized defect tracking and a unified build process: these decrease the turn around time of the corresponding tasks, as individuals do not need to explain anew the same process to report defects, the same location where to find information about defects, and the same procedure to derive one particular version of the software artefacts that constitute the system. As shown in Figure 3-7, setup time for all these tasks is critical, and in the absence of standardization, individuals will spend time extracting useful and useable information from their counterparties. Similarly, artefacts must be tagged appropriately in source control to facilitate software archaeology tasks (these are what software engineers do when they comb through revision control history to determine what has changed, this is how they pinpoint a specific change that caused one particular mode of failure).

As mentioned in Section 3.4.1, another task related to standardization is *triage* of bugs: this task is performed by engineers to indicate severity of incoming failures and defects, remove any duplicates, steer towards the correct group for actual fixing or documentation, or select which mainline or mainlines the defects will be fixed on. Typically, the organization will strive to fix the least possible amount of problems on its oldest mainlines, as it incentivizes customers to upgrade to newer fielded versions of the product, and decreases risk on fielded mainlines.

Section 2.7 hinted that software quality measures could be monitored, but metrics are notoriously hard to port between projects or companies: in the context of large-scale software engineering of product lines, it seems that these should be easier to establish, as successive release cycles are comparable. The organization can accumulate historical measures and compare current measures to past in a more reliable way.

3.2.1 Builds

As noted in Section 2.2, building the product occurs at many levels within the organization: individual engineers build it to be able to develop new features or fix defects, release engineers

One product line I worked on did not have standardized build processes, getting a new hire started and able create debug builds would take about a week of work to set things up just right (and each individual would have to re-discover these things for himself!). Another product line had a very standardized environment, and one could get a build environment started in minutes. I believe that the value of making this obvious and rapid cannot be understated. A side-effect of these two modes was that in one environment, engineers used few build workspaces and mixed changes, and had to tell the source control system specifically what had changed (thereby leaving room for error), whereas in the standardized environment, engineers would create many build workspaces, one for each individual change and let the source control system figure out what had changed, which left much less opportunities for mistakes.

Figure 3-7: Example of problems when builds are not standardized.

produces official builds shipped to customers, and possibly in a hierarchical organization, release engineers will also be organized around each shared asset and each product, each of which will see an official *internal* build set up. Each level has its own rhythm: some build weekly, others daily, others yet multiple times per day. [Dikel, Kane & Wilson 2001, p. 92] says one could allow for breakage and use CM[1] to keep on working. This trades release engineering work for work on the developer's part to keep their own environment stable enough to be able to work. Following industry practice, we argue here that a *no broken windows* policy and a single Best-So-Far CM view is better as it fixes broken builds as early as possible, when a minimal number of developers are involved. When this policy is not respected, breakage occurs constantly and masks other issues, engineers have to perform extra setup to get a working version of the system, time is wasted by all and multiple occurrences compound to bring the software production system to a complete stop.

Some organizations go so far as to associate some social stigma to *breaking the build*, as a way to incentivize individuals to be specially careful not to break other's builds. This has shown to be working well in many organizations, but does not work 100% of the time, for the occasional human error occurs, as unpredictible bizarre system-wide interactions do. In summary, as McConnell notes [McConnell 1997, p. 206]:

> *The project team brings the software to a known good state and then keeps it there. The software is simply not allowed to deteriorate to the point where time-consuming quality problems can occur.*

[1]abbreviation for *Configuration Management.*

3.2.2 Source Control

In single-person development, one can work with a single mainline, possibly with no branching at all, as shown on Figure 3-8, which depicts the timeline of events.

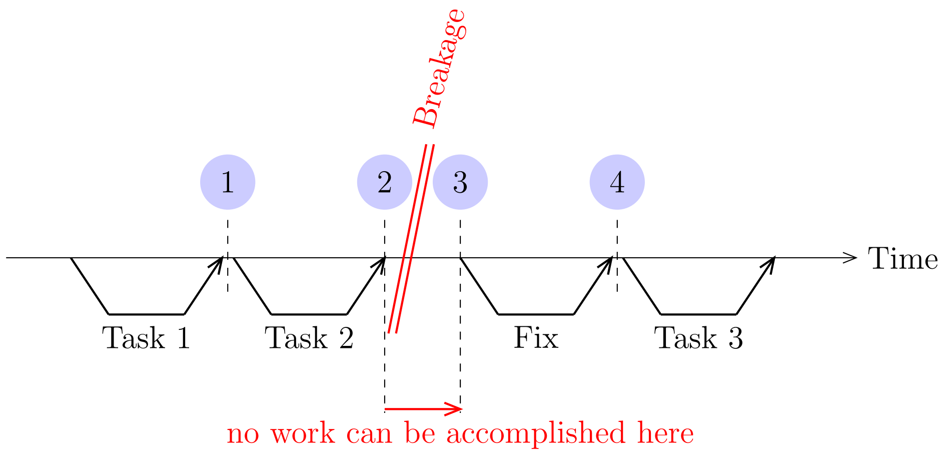


Figure 3-8: This shows how source control is used with a single mainline by a single developer.

Tasks are performed in sequence:

– The developer *checks out* the source code, and performs the work for task 1. When this work is completed, the code changes are submitted to the source control system at time 1.

– The developer *checks out* again the *newest* revision of the source code files, and performs the work for task 2, submitting changes at time 2. At this time, some *breakage* occurs: this could be caused by a syntax error in the source code file, a change in runtime behavior that is seen as a unit test failure, an incompatibility with some other change, etc...

– At time 3, the engineer starts developing a fix for the problem he caused, which he submits at time 4.

– After 4, new work can be started, here shown as Task 3.

This simplified example shows that even in single-person development, one needs to be careful to checkin small chunks of changes, as checking in one large change makes it more difficult later to pinpoint a single atom of change that caused a problem. Similarly, having a suite of unit tests and running them assiduously may sound like a lot of unnecessary work, but it in fact is a manual smoke test that can help detect regressions early. Figure 3-9 shows what happens when a single developer needs to work on two tasks in parallel, note how that is exactly the same as what happens when two developers are working on the same branch concurrently: one person submitting a change that breaks the build paralyzes the other person trying to get work done. This situation is avoided by either dividing labor such that no two people work on the same portion of the system at the same time, or by using branching to isolate workers from each other.

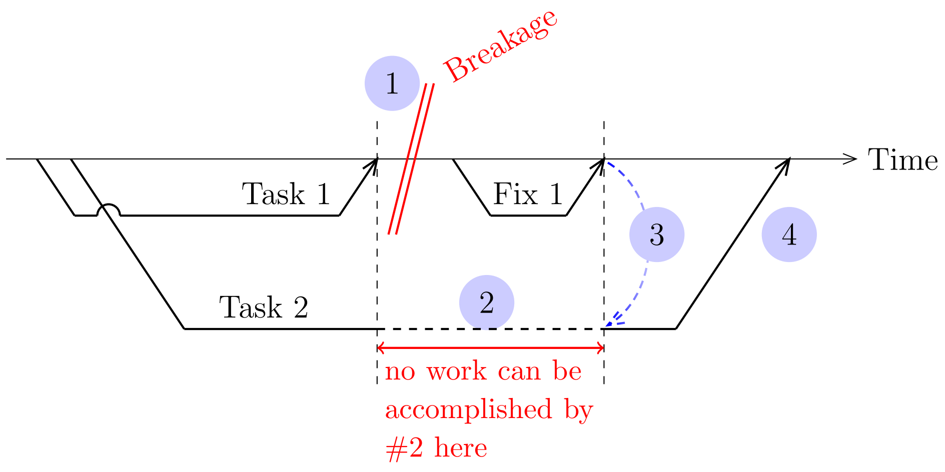


Figure 3-9: This shows how source control is used to perform two separate tasks on a single branch

Figure 3-9 shows what happens when breakage occurs:

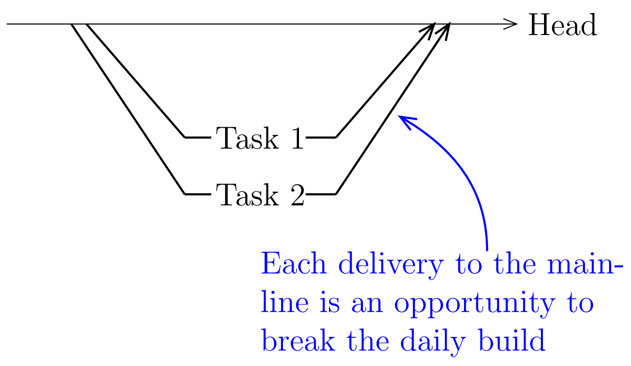
– 1 is the *break the build* event,

– 2 is the period of time the second developer can neither synchronize (because doing so would break his build, preventing him from working) nor promote (since he needs to synchronize first),

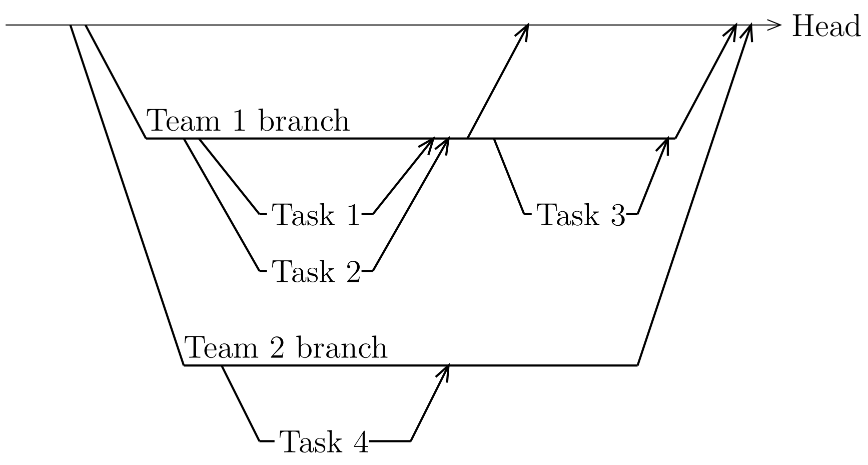
– 3 is the first moment when the second developer can synchronize to pick up valid changes that will not break his build

– 4 is the moment when the second developer can actually submit his set of changes. Note how long he has been blocked from doing this.

In many-engineers development, the situation highlighed in Figure 3-9 is magnified, and therefore each developer must be able to create a private workspace to experiment with changes without interfering with the work of others, yet each must also have the ability to publish changes to others when the change has reached sufficient maturity. The *head* or *Best So Far* version of the mainline represents the accumulation of changes whose sum is the current state of the software system, from which release engineering staff can produce a master build that will eventually be shipped to customers.


Head
Task 1
Task 2
Each delivery to the mainline is an opportunity to break the daily build

(a) Small company, single branch



(b) Large company, single branch with hierarchical isolation

Figure 3-10: Scaling from single-team development to multiple-teams development process.

Deciding how many mainlines to have and support is a managerial choice, where the decision makers must balance the needs of customers to have many such active mainlines with the costs for the enterprise, which decrease with the number of mainlines. This decision is called the *branching discipline* of the company. As the number of engineers, teams or customers grows, there will be a tendency to create new mainlines indiscriminately, this tendency must be managed in accordance with company objectives. [Berczuk 2002] formalizes the various possible policies and branching disciplines, in the form of *patterns*. Similarly, as the company grows, there will be forces inside the company that will attempt to either create shadow mainlines, or even parallel repositories: yielding to these forces can spend enormous administrative resources just to manage those mainlines. Such fragmentation can be the result of acquisitions or, as highlighed in [Bosch 2000, p. 287], the result of internal infighting following divergence of strategic needs of various products: indeed some teams may *leave the product line officially, or practically*, doing so by forking the software architecture, moving to private branches within the source control repository, or even establishing a completely separate source control repository and developing completely outside of the official software production system.

Therefore one can see that using a single source control system is a powerful unifying force within the software development organization. It avoids the local creation of new variants to fit one team's immediate product needs, as that phenomenon makes it harder to converge later, and makes products within the product family incompatible one with the other. The flip side of this is that experience seems to show that software organizations tend to need to experiment on the side, and when forbidden to do so, will find ways to do it anyway: so it often happens that the master source control repository is peppered with half-finished prototypes, which do not contribute to the product line, which may not even be visible to customers, but are still part of the software production system, with associated administrative costs in building, computing metrics, etc..

3.3 Physical structure of large software systems

While small software products can live without proper structure when there are only a few software artefacts which participate in the building of the end product, as product size increases, the architecture must get realized in a physical structure such that:

– it be possible to rebuild portions of the system without having to rebuild the whole system,
– it be possible to rebuild incrementally the system. Builds from scratch can take hours, if not days, and quickly make it unfeasible to rebuild the system,
– parts can be swapped in and out of a pre-built image of the system. This helps with debugging since it can be performed in a live installation, and not just in a debug setup on a developer machine.

In this section, we will explore a few of the ways that architecting can help when project scale increases.

3.3.1 Modularity as the support for structure

Modularity, as explained in section 2.8, is the decomposition of the system into subsystems and components-within-subsystems such that these parts can be worked on, modified, and substituted independently. [Lakos 1996] and [Parnas 2001] before him have explained the great benefits that architecting in decoupled modules brings. In software, structure is the abstract division of the system into modules, it is realized physically in the physical layout of modules in the source control repository. The developer typically checks out one or several modules onto his machine in a workspace, builds them and then assembles a complete system from his private build and modules from the official release engineering build: it is very rare for the individual to check out and build the entire system. Modularity and decoupling are the principles that allow the developer to work with a partial private build layered on top of a pre-built image of the system.

3.3.2 Conway's Law

McCormack & al. have demonstrated that *products tend to mirror the architectures of the organizations in which they are developed* [MacCormack, Baldwin, & Rusnak 20121], therefore a hierarchically decomposed system architecture will tend to be produced by an organization which is layed out in similar fashion. This is merely a reformulation of Conway's law [Conway 1968], which Conway phrased as *Systems Image Their Design Groups*. Not only is this a fact observed in the wild, McCormack argues that this is a desirable property. We carry this argument further and want to demonstrate that this division of labor in fact presents enormous leverage for managers.

One hypothesis of the present work is that having a hierarchically contained checkin process improves quality: it improves the availability of "good" builds, and it allows QA to find errors faster, thereby leaving less undiscovered errors in the product. The underlying assumption is that a rhythmic pattern of delivery is desirable. In other words, stability is better, the rhythm of the production system aims at avoiding stalls due to human error, yielding a predictably good output.

Cusumano and Selby have documented how Microsoft does this in [Cusumano & Selby 1997]: using small teams with frequent synchronizations. This *synch-and-stabilize* approach allows to isolate developer teams from each other, giving each a stable environment to work in. The module structure of a system can be engineered or architected in, but even systems that were not engineered according to modularization principles manifest such a structure [MacCormack 2006], or can be modified a posteriori to have such a structure [Akaikine & MacCormack 2010].

3.3.3 Build Times

As soon as the product codebase becomes large, it become impractical for individual engineers to have to rebuild the whole system (even components they do not care about when working on one particular issue). The location on disk where engineers can perform private builds is called a *workspace*, or alternatively a *check out directory*, a *working copy* or a *view*. Requiring a full rebuild when setting up a workspace increases the basic overhead for software engineers. Setting up should essentially be free.

Compile and link-edit times will increase super-linearly if not engineered carefully: [Lakos 1996, p. 87] gives details for large C++ systems, large Java programs such as NetBeans or Eclipse use the same practices to decompose into smaller chunks and build very large systems. Field deployment costs similarly increase in monolithic systems: this alone is a strong incentive to modularize. Although this is scarcely documented in the academic literature, very large scale software production systems have resorted to some caching scheme to avoid re-compiling the same exact revision of software artefacts many times over, to avoid being limited by compile times or relink times. This matches our experience in industry, where release engineering produces one company-wide official build daily, each team in turn does so, and individuals build their own partial build layered on top of the previous two: this significantly decreases the time to get a working build with the latest assemblage of software source code, and allows engineers to set up within minutes, even with full system build times in the hours. Examples of such build optimization schemes are the *distcc*, *ccache* or *Vesta* systems.

As system build times soar, setup time for the engineer working on a defect increases, which diminishes the overall capacity for propagating fixes: fixing one defect typically takes a lot of debugging, researching, understanding, but merging a known fix onto other branches does not have this overhead, and the time for this operation is purely proportional to setup time and testing time. In other words, after fixing the defect on one release, when the corrective change is identical on other releases, there is a fixed cost of testing and administrative overhead just to submit the change on each release where one touches the code.

3.4 Product Lines

One major challenge brought about by the scale of the engineering endeavour is that effort grows in a fashion that is worse than linear in the size of the software product. In an attempt to manage this scaling issue, software development companies resort to using *Product Line Engineering*. This is a discipline of systematic engineering of the members of the product line.

When one considers the context of product line evolution, a software vendor typically produces a product family or product line (sometimes consisting of a single product), which evolves over time. In such a setup, the organization selects a release pace (once per year, twice per year, or as frequently as multiple times a day in certain organizations), thereby time-boxing each release.

Product line engineering presents several differences when compared to engineering within a single project:

– Projects are often one-of-a-kind, or use state-of-the-art technology. Long-term product line engineering develops many products similar if not in features, usually in the use of technology: instead of a one-time use, it is repeated use over a long period of time, and therefore sees learning effects. The economic benefits come from the repeated reuse of a set of components.
– The above makes projects risky, whereas long-term product line engineering can hedge risk by discarding untried or unsuccessful technologies.
– Technology development happens in the background, possibly outside the organization, and the development organization uses technologies it has tried and approved.
– Short-term effects have long-term consequences, therefore effects may not be visible within the relatively shorted time horizon of a single project, whereas long-term product line engineering will inevitably incur the long-term effects. Using product line engineering has initial costs that are recovered over the long term by low-cost reuse of the components selected to be part of the product line architecture.
– Long-term consequences may be either beneficial or detrimental to the success of the engineering effort, when the effects of feedback seem bad in the short-term but turn out to be good in the long-term, this is called the *worse before better* effect. This effect that is often seen may cause the cancellation of a project, whereas in managing over the long term it should be expected and accepted.
– Short-lived projects can assume that there is no turnover, as the manager can always try to convince a worker to “stay till the end of this project”, but for long-lived development efforts, turnover will happen and must be accounted for.

These make product line engineering management substantially different than project management.

3.4.1 Planning and Execution of Product Line Engineering

The organization essentially runs the simplified process detailed in Figure 3-11: all states are executed concurrently and repeatedly, but each new release goes through the states described in the figure, with some tight iterations due to bug-fixing, termed *firefighting* in cases of emergencies, that is those cases where a defect is found in the field while customers are using the system in a production capacity, and therefore require immediate attention.

For instance, Release 12 could be in the *Design* stage, while Release 11 would be in the *Build* stage, and Release would be in the *Beta testing* stage (also called *Certification*), and Releases 9, 8, ... would have been deployed in the *Field* with customers. In the rest of this document, we shall call our releases $N-k, ...N-1, N, N+1$. Release N is the *active* one, in other words it is the release currently under development.

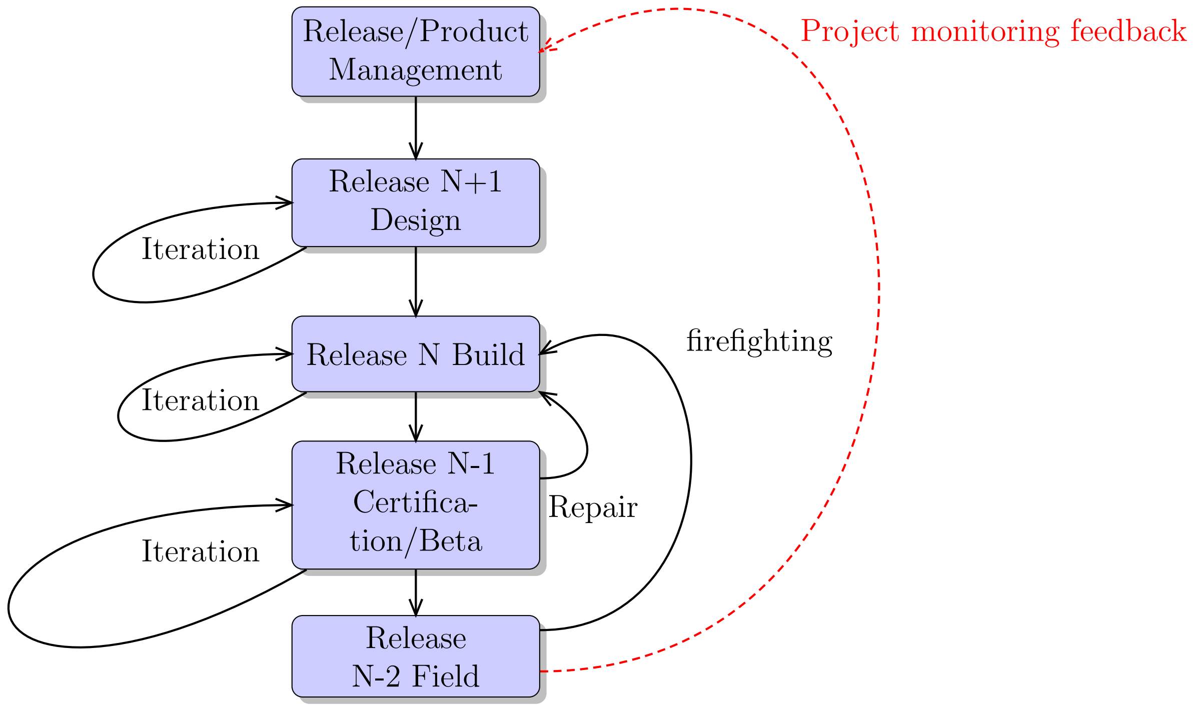


Figure 3-11: The phases of development for long-term release-based product evolution.

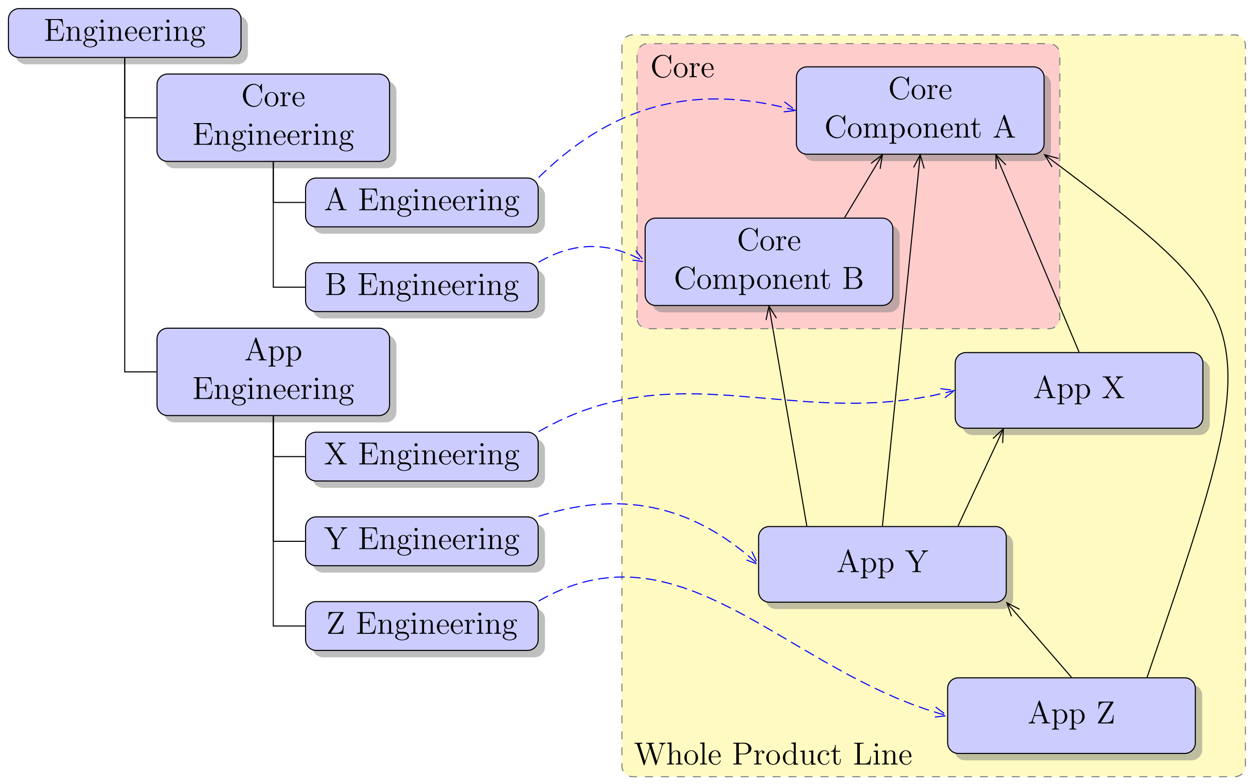
In Figure 3-11, we may note that each of those phases executes concurrently, and any one release goes through the phases in order. The red arrow indicates a feedback loop for organizational learning. A more complete process might include:

- market research, which focuses both purchasing agents and end-users. The word "customer" here is ambiguous as it can be used to mean either.
- more extensive planning and scheduling sectors.
- an expanded quality assurance sector [Abdel-Hamid & Madnick 1991, p. 70], including the multi-level support function which acts as a filter to triage problems reported by customers. This operation is about classifying faults and prioritizing them, detecting which faults are in fact defects and which are duplicates, explaining misuses to end-users, detecting design problems that require additional rework.
- a fielding and deployment process, which may require staff to visit customer locations to help them configure and upgrade systems, as well as help IT departments migrate away from legacy systems. Such roll-outs may become large projects onto themselves, and require much testing by customers to ensure continued availability of mission-critical systems.

These processes, as well as feedback loops they contribute to, are beyond the scope of this thesis: more extensive modeling work should be a very interesting and fruitful topic of future study.

3.4.2 Definition of Feature Graph

In software product line engineering, the product architecture is logically decomposed into features, each of which represent a logical unit of functionality for the user. Each product within the product line architecture expresses which features it includes: this is the mechanism that allows reuse of features by members of the product family, and therefore features are often arranged in a product-feature matrix. However, such a matrix is not sufficient to express that in addition a feature can depend on other features: as a consequence, features are also arranged as a feature graph, sometimes called a dependency graph, as in Figure 3-12b.



(a) Sample organizational hierarchy, with two sub-levels within the core, and three products.

(b) Sample component dependency graph for a product line engineered in the organization from 3-12a.

Figure 3-12: This shows how components are stored in the repository on the left, and dependencies between components on the right. Architecture diagrams produced by the developers often look like block diagrams, and indeed such diagrams do show dependency information.

If we refer back to Conway's Law (Section 3.3.2), experience has shown us that the organization which produces a product line architecture like Figure 3-12b is itself organized as in Figure 3-12a, with the correspondance between the two hierarchies shown by dashed arrows from the left figure into the right figure. This is important because the dependency links that represent edges of the feature graph become interfaces between components of the software production system. In other words, the teams that produce each of the components have interfaces to their clients: there are contracts between those organizational units, those contracts relate to the schedule of promised deliveries, as well as to the stability of the deliverables in the light of evolution over many release cycles.

3.4.3 Variability Management

Product line engineering is largely about what the organization needs to do to manage *variability*, which is the variation of artefacts depending on the context in which they are reused. [Pohl, Böckle, & van der Linden 2010] explains the difference between variability in space and variability in time:

– *Variability in space is the existence of an artefact in different shapes at the same time.* as defined in [Pohl, Böckle, & van der Linden 2010, p. 66]. This is about the shape of the product family: what product exists, which variants are available to customers, or exist entirely within the design space of the software engineers. Note that space can often mean memory usage of a software product; this is particularly in the embedded products space which favor build-time variability as opposed to run-time variability, since we do not know these well, we will not not expand on them any further here.
– *Variability in time is the existence of different versions of an artefact that are valid at different times.* as defined in [Pohl, Böckle, & van der Linden 2010, p. 65]. This is the variability we intend to look at in this thesis.

3.4.4 Use of variability to create products within the Product Line

Once a software architecture is in place for the product line, when the company decides to create a new product, it needs to *instanciate* the product line architecture into a product architecture. This phase of Product Line Engineering is about deciding how the new product will be built, what components will be reused (or *instanciated*), how (that is, how will variability be bound in this specific product, such that the component presents the required features with the specified functional or technical attributes).

Once variability has been designed into components, it is up to the product manager of each product member of the product family to decide how to bind the variability to the specifics of his particular product. The time when variability is bound is also important to the architecture of the software production system:

– it can be bound at build time. During the builds performed by release engineers, the instantiation of the reused components includes the information necessary to bind variability. For instance, some configuration can be specified through compile-time constants, which can for instance change the static size of structures or enable/disable large portions of the code. This is crucial because this makes it difficult to include multiple instanciations of the same component within the same build, which in turn forces release engineering to generate one build per product. This can increase administrative overhead and increase the number of variants of the code that exist in various members of the product family. When this is used, there will be a tendency for users of a component to clone the component locally when new features or alterations required by the product family member are not performed in speedy fashion by the provider of the component.

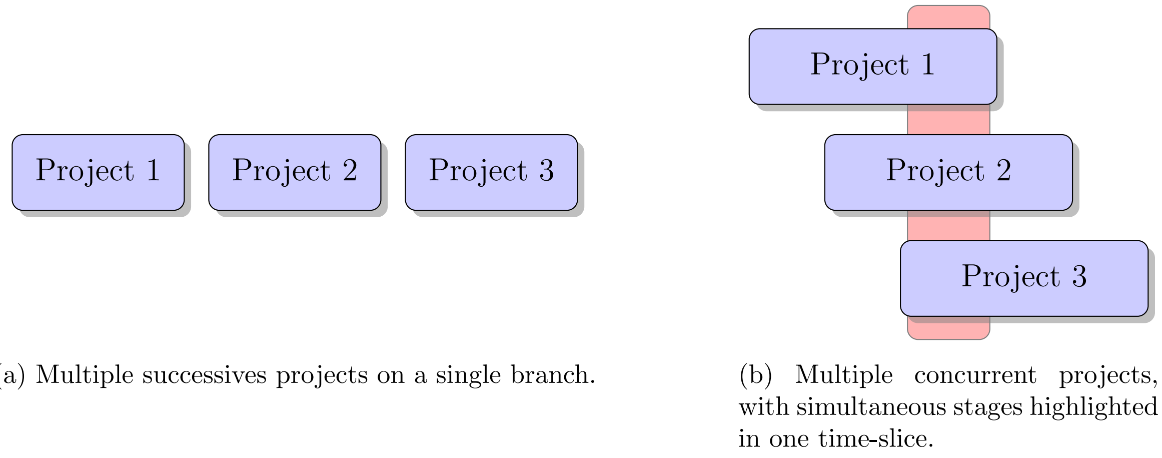
– it can be bound at run time. This mode of operation gives the greatest benefits, as a single build produces one software artefact that can be reused in multiple products simultaneously. The advantage of this mode of operation is that it significantly decreases the number of active branches in which one component exists. This kind of variability may be more difficult to achieve, but it can help make the administrative overhead in release engineering a fixed cost as the number of reuses grows.

Another point is that software development organizations almost always face a tension between being a product organization and being a service organization, and even product organizations often possess elements of service organizations in the way they function. For instance, because of the financial leverage of customers, the company can elect to support maintenance branches for as long as sufficient funding is present or as long as a customer requires it (contractually or through their financial leverage). An example of this is the long life of the Windows® XP product from Microsoft®.

This intrinsic tension tends to induce the organization to attempt to produce more variability points, more variants and more custom builds of products. It is up to the college of product managers and to the product line architect to balance those strategic needs with the long-term well-being of both the product line architecture and the organization.

3.4.5 Scheduling within the Product Line

The Product Line Engineering literature is concerned with a static view that defines what the product line architecture looks like and how can member of the product line be created. In the present thesis, what concerns us is the dynamic behavior of the product line and the organization that maintains it over time. In particular, the organization usually produces many successive releases of the product family, or more precisely of the member of the product family (the product family is a theoretical concept, which customers never actually see: they only sees the realization of the family concept into specific members of the family).



(a) Multiple successives projects on a single branch.

(b) Multiple concurrent projects, with simultaneous stages highlighted in one time-slice.

Figure 3-13: Difference between sequential and concurrent projects

As depicted in Figure 3-13, each ongoing release branch (sometimes called a *mainline* or a *baseline*: simple or small projects usually have one of those designed *head* or *trunk* or *Best-So-Far*, complex projects usually have several) goes through the same phases of:

- Requirement
- Design
- Implementation (sometimes called *coding* in the software development sphere)
- Test and stabilization, sometimes called QA (Quality Assurance), QE (Quality Engineering), or more simply *test* or *beta*.
- Maintenance, or in-the-field phase, which ends when that release is discontinued or EOL'ed (*EOL* is short for *End Of Life*, a common acronym in the software industry to mean the retiring of a product).

Release branches are labeled, usually numbered in sequence, as experience shows that using any other kind of label makes it difficult to explain to salespeople and customers alike which release is the "latest" or the "best" to have. Release branches are sometimes arranged in what is called the *release train* model: each branch is a *train*, scheduled to start at a particular time, and to arrive (i.e. finish) at a specific time. We will assume that at any point in time there is only one release undergoing development work, others are either in the planning phases (future releases), or in test and maintenance phases (past releases). Releases enter and leave phases in staggered fashion: when release N is shipped, N+1 is moving into beta-test stage, and N-1...N-k all slide down in focus level. Next, when release N leaves beta-test stage and enters general release stage (i.e. a *Golden Master* is generated, to employ industry jargon, and shipped to customers for deployment), development teams stop active work on release N and move on to working on release N+1.

Managers tend to view successive releases as sequential projects, as displayed in Figure 3-13a, whereas engineers in the trenches working on those releases think of them differently: they tend to view those release trains as concurrent, as shown on Figure 3-13b. As will be explained in later sections, not only are those releases happening concurrently, they also compete for the same resources.

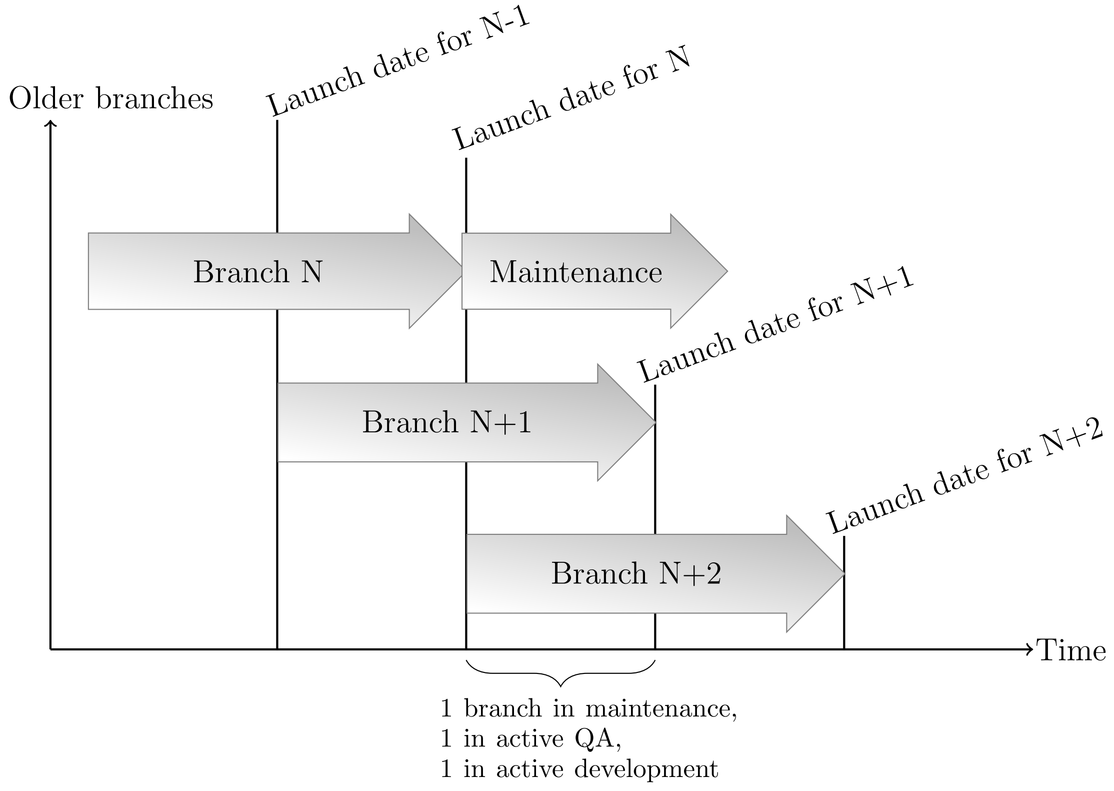


Figure 3-14: Multiple mainlines active at the same time, similar to Repenning Model-Year example.

Note that selecting how many mainlines to keep active is a matter of policy decided by management. It is a powerful lever since having too many mainlines active can easily swamp the development teams, which then have to do firefighting on old mainlines. Typically, the number of mainlines kept active depends on many the organization can handle in terms of resources, but it can also depend on the funding that customers can provide to keep their own private mainline active (This is sometimes the case with automotive and aerospace customers, which tend to do their own product development with *programs* that last several years, and changing/upgrading software midstream is difficult or risky for them). Another point is that different customers put pressure on different branches, each on their own, pulling the R&D organization in different directions, and therefore the organization needs to balance the needs and wants of customers with its own strategic and tactical goals.

Some organizations will have a policy of employing separate staff for maintenance work on old mainlines, but usually it is difficult to compartimentalize work completely, and the maintenance staff will need to consult with the original developers, which has non-linear effects on the time to resolution of defects on the concerned mainline, but also induces more interruptions, more meetings and more demands on current mainline developers.

Another point to consider is that others have demonstrated that complexity has a cost [Sturtevant 2013], and structure has an impact on development ([Sturtevant 2013]) as well as on maintenance costs [Akaikine & MacCormack 2010]. These works prove that architecting the product such that its structure respects the principle of hierarchy and modularity brings tremendous value.

It is important that other stakeholders do not unintentionnally create or maintain activity on old mainlines, when creating new products or designing solutions for customers. Mainlines are like regularly-scheduled trains, and stakeholders must pick in advance which one they are going to get on. This is especially true when the company has a professional services arm, as it is then subject to the intrinsic tension between products and services. Here we are talking about companies that are product companies, but may perform services (or have a partner perform services) accompanying the product purchased by the customer. For the product side of the business to remain the focus of the company, one should be careful not to let services cause additional work to the product side. Similarly, within the development organization, it is paramount that no part of the organization duplicates the components owned by another (this is documented as the Cloning pattern in [Dikel, Kane & Wilson 2001]).

A point solution is a solution that fixes an immediate issue without regard for the longer term; in the present context, such solutions, including the Cloning one mentioned above, are tactical resolution of urgent customer issues without regard for the policies that drive product line development and strategy.

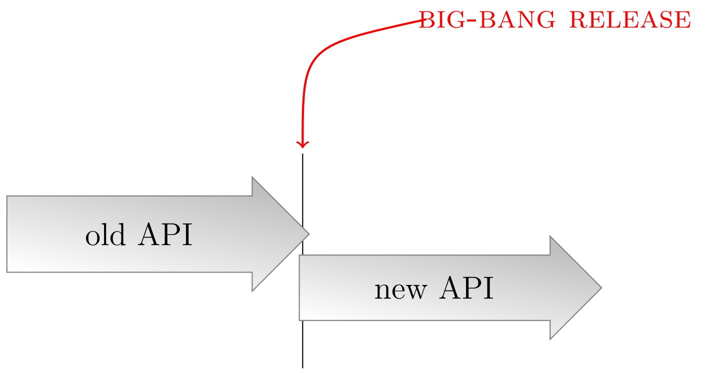
Point solutions are a perfectly fine use of the product line, but the organization must be careful to only use point solutions appropriately and to discriminate features that really belong in the product line, which may require post-poning delivery (hence the tension with the customer-focused services branch of the company).

3.4.6 Stability of Product Line artefacts

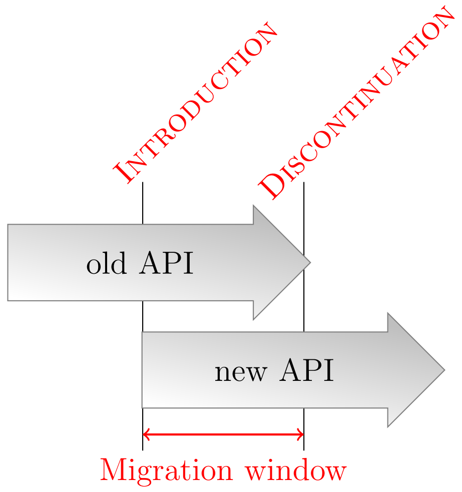
One example of additional cost due to scale of the system is what happens when an engineer needs to change some component-level API. An *API* is an *Application Programming Interface*, it is the programmatic interface that one component presents to its callers. It is an interface in the systems architecture sense. Software interfaces are a much more general concept than programming interfaces, for instance fileformats and wire protocols are in fact “interfaces” in this sense, though not in the programming language sense. Even at the programming language level, “interfaces” do not need to be language interfaces: in fact (and this is very confusing to many), for example in Java, component interfaces should often explicitly *not* be Java interfaces, but final classes! (See [Tulach 2012]). Here is what happens when an API must be modified:

– In single-person development, the engineer can change the definition of the API and all its occurrences in a single delivery in the source control system. There is a single point in time where the API ceases to exist. This is shown in Figure 3-15a below.
– In large-scale development, one needs to introduce a migration path and a window of time for callers to perform this migration in their own source code: this means adding the new API without removing the old one, deprecating the old one, then advertising to all callers that they need to migrate and start using the new API. After a migration period, the engineer may finally remove the old method. This may seem like more work overall than in single-person development, but this ensures stability of the system to its dependents and allows to change moving parts in flight. This is shown in Figure 3-15b below.
– Sometimes the old API and the new one cannot co-exist at the same time. An example is given in Figure 2-7; in that particular instance, a reversal of the change which caused the issue had to be made visible to all developers at the same time. The system should strive to make it possible to preserve old APIs while new ones are introduced, this is sometimes done by isolating the two APIs from one another in some sort of container such that only one is visible to callers at any time, and callers have to explicitly opt in to migrate to the new API.

This is typically the case for programmatic interfaces, but also happens for interfaces in the more general sense of the word: file formats and wire protocols also fall in the category of interfaces that must be evolved in a backwards-compatible way.



(a) In single-person development, APIs can change at will.



(b) In large-scale development, one needs to introduce a migration window.

Figure 3-15: API changing over time

Further, if the boundary of the whole-product system extend outside the enterprise (for instance, if the enterprise publishes APIs and partners or customers build additional applications or extensions layered on top of the system delivered by the enterprise), then one must take special care to enforce those migration windows to give time to partners to migrate. This means that the migration windows typically have to span more than one release cycle, as one does not want customers do discover their application is broken after installing an upgrade provided by the software provider. Assuring backwards compatibility, even temporarily, builds trust in the customer base, it advertises to customers that the producer cares about preserving their investment in the product system, one effect that is much desired for the long-term viability of the software product.

One experience I had with having multiple active branches of the same product family is very telling: we had a new capability to implement that was estimated to be such a large change that the team collectively thought it could not be done in the timeframe of a single release. We then implement the new capability the simplest way possible in release N with a master switch to disable it, and then went back and implemented it a second time differently in release N+1. This tactical solution allowed to roll out the capability in a released product on the market as early as possible, while the strategic implementation on release N+1 learned from the initial implementation and was able to bring important benefits such as performance and scalability that were much more complex and time-consuming to engineer than the simple solution.

This goes to show that managerial choices can appear to increase the workload on the production system, but sometimes doing more work is beneficial in terms of breathing room, time-to-market, or simply expect to have to learn as you go what works and what does not. As Frederick Brooks highlighted in [Brooks 1975],

Plan to throw one away; you will, anyhow.

Another lesson from this example is that when the expected size of a new capability does not fit in one release's timespan, adding some of the components to release N, even if not active or de-activable by a master switch (generally one not advertised to customers) is a worthwhile mitigation measure. It allows other stakeholders within the code production system to start working with the new capability even if it is not yet fully active. This strategy also prevents developers from accumulating massive code changes and delivering them all at once at the start of one release. One such large delivery acts as a step function on the system, usually an event with bad consequences in a non-linear system with feedback loops.

This is an important lever to allow others to start their work when the new feature requires changes in many components.

Chapter 4

Challenges of large-scale software engineering

In this chapter, we want to examine a set of issues unknown to small-scale software engineering organizations:

- **Section 4.1** examines what additional issues occur when a software development effort grows to span multiple sites. Here we want to find out if and how the inevitable latencies influence system behavior.

- **Section 4.2** compares two different modes of growth: the first one is natural growth occurring by growing headcount, the second one is external growth, or growth by acquisition of other entities. We want to show here that these modes are different and induce different dynamics in the system of the enterprise.

- **Section 4.3** looks at the specificities of products that incorporate content from third party vendors, which is the case of most software product of any significant size. This aspect is significant because third-party components typically have a separate life cycle, and dependencies that cross the boundary of the organizational system to reach outside create feedback loops with a long delay, which therefore require special attention.

- **Section 4.4** looks at software product lines and what benefits and drawbacks these brings to large scale software development. A *software product line* is defined by the Software Engineering Institute [SEI 2013] as

  > *a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*

Recognizing and managing these assets is valuable to the engineering manager, in the present work, we want to examine how managers can indeed use this knowledge to their advantage. It is important to highlight here that managing a portfolio of applications, while a perfectly valid way to scale the number of software applications produced by the organization, is not a focus of the present work. In addition, we restrict ourselves to pure software systems, such as Microsoft® Office®, the ones we are most familiar with, as systems termed as software-intensive can often include non-software parts which bring additional complexity.

4.1 Scaling from single-site development to multi-site development

There exists a point in time where any development organization will outgrow the original building, city and eventually country it started in. This is a consequence of globalization, and leads large companies to conduct *Global Product Development* [Stark 2000]. This is unfortunate, as highlighed by [Allen 1984, ch. 8], but necessary after a certain size has been achieved, if only because the locally-available prospective employee pool has been exhausted. Often this happens because of acquisition of other organizations that could be located so far as to make relocation impractical. In addition, in the internet age, more and more employees also have been accustomed to working remotely and expect the ability to do so as a condition of working for an employer.

4.1.1 Latency caused by distributed work

The issue here is that working in a geographically dispersed fashion, whether alone (maybe from a home office) or in group (maybe from a office remote from the enterprise's headquarters), adds delays into the software delivery system of the company. Developers synchronizing their workspaces from remote locations are bound by the capacity of their internet connections. Whether one uses VCS, *Centralized Version Control System*, or DVCS, *Decentralized Version Control System*, there is the associated cost of transferring over the wire not only the actual source code changes as deltas, but also the version history of the files. Figure 4-1 shows a diagram of hierarchical delivery system and pinpoints in red the communication channels with latency between sites.

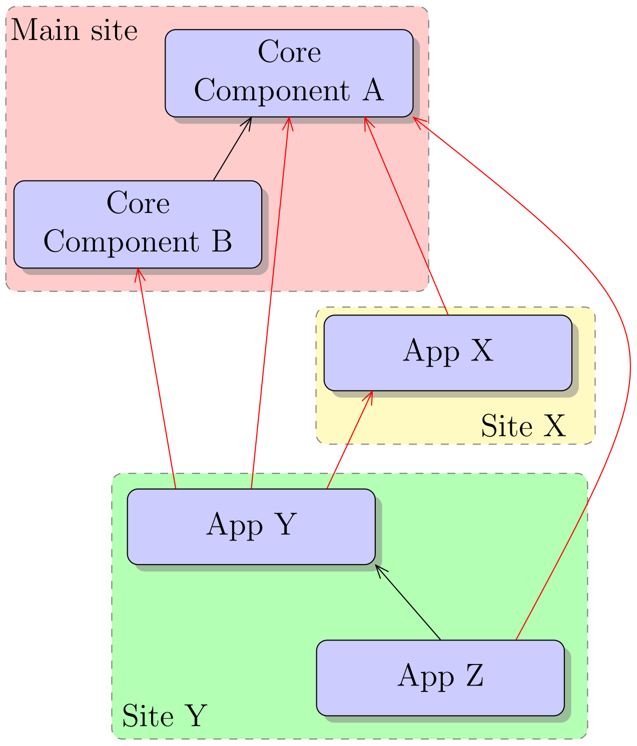


Figure 4-1: Example in the context of Figure 3-12b.

Similarly, as the daily master system build produced by *release engineering* at the main site needs to be propagated to remote sites, some latency is incurred on those sites: they have an opportunity to opt out of the propagation of a broken build, but if they do get one, co-located engineers will not be able to work that day. One way to avoid this effect is to propagate build with lesser frequency, but that needs to be balanced by the need to do system integration as early and as often as possible, otherwise rework can be discovered with a long delay, increasing the duration of negative effects of rework.

4.1.2 The value of co-location

[Allen 1984] demonstrated the value of co-location of knowledge workers, showing that communication decays significantly with distance. Unfortunately in today's world, geographical dispersion is a necessity, as companies are formed or evolve to span multiple locations. Such dispersion does bring some benefits, such as the ability to have on-call personnel around the clock, or the capability to test large software systems in realistic deployment scenarios where multiple nodes are distributed in multiple countries and timezones. In the context of a daily master system build, it is convenient to delegate smoke testing to elements in the organization who are located many timezones away and whose workday spans the nighttime of the development team: this makes results available at the beginning of the next workday.

4.1.3 Organizing around multiple locations

When the teams at each location are independent of each other, the resulting work product is not really considered a system, but merely a collection of products bundled together. The next stage in the evolution of the company is to manage the work products of remote locations almost like third-party vendor-provided products: this is possible, and indeed upon acquisitions and mergers, that is typically how work is initially organized. However, such an organization does not allow for much integration between products, and delay system testing as the operation of assembling the products together is done after release engineering on each site has produced its master build. It is much better to have the ability to produce a master build of the complete collection of systems in one location, but require more and more frequent coordination to be able to produce a periodic build (a daily build is best, some organizations choose a different period to operate on).

[Bosch 2000] makes the point that

> *geographical distribution of the teams developing the products in the product line may cause a company to select the domain engineering unit model because it focuses the communication between the domain engineering unit and each product engineering unit, rather than the n-to-n communication required when using the business unit model*

and this can be generalized to the *hierarchical domain engineering unit*: the leaders should be careful to distribute work among sites so as to not create cycles that span channels of communication with latency. In other words, one does not want to locate work on *core* component in a remote location other than the location of the central repository and master build. To do so would risk that regressions in *core* components would be detected late and repair would also incur delays: this is a typically feedback loop with a delay, we could characterize it as a rework cycle with a delay caused not only by the intrinsic time to discover defects in the work product, but also an additional delay built into the software delivery system.

4.2 Organic growth vs external growth

4.2.1 Organic growth

Organic growth is achieved when the organization grows naturally, that is, it increases head-count at a pace such that new hires can be absorbed into the system without disturbing it too much. As communication needs to grow quadratically with the number of players if everyone communicates with everyone else, organization typically use some partitioning to diminish those undesirable effects and decrease the amount of required coordination. The partitioning used by the overwhelming majority of organization is hierarchical. Even in the presence of such hierarchies, knowledge workers intrinsically know each new hire will endure some incompressible assimilation time, which may very well vary from one individual to the other.

4.2.2 External growth

Many large commercial enterprises grow by acquiring other companies. The case of a software product company acquiring another is specially interesting in two regards: in regards to the acquisition and what leads to it, and also in regards to the merger part of the process. In this work, the more interesting point is what happens when merging has been decided and actually happens. The organization is growing and absorbing a nimbler entity: this is quite different from natural growth, and represents a sudden step in size of the organization. The respective sizes of the two parties involved in an acquisition event followed by a merger has an impact on how events unfold in the aftermath of the legal merger. The legal counsels of the two parties draft a merger agreement which maps out a schedule for the legally-defined merger event to happen; usually this formal date dates the formation of the new entity, but much work remains to be done within the entity for true and complete integration to be finalized. The larger the organizations, the more steps to be taken to fully realize the acquisition and/or merger operation.

4.2.2.1 Acquisitions

The enterprise may elect to acquire other companies:

- to acquire market share in new markets. This may be done to acquire a new application to add to the portfolio, maybe to enter a new adjacent market. This could also be done to acquire a brand name, or to increase presence in some geographical market.
- to increase market share in a market it already is present in. In this case, the enterprise may own several competiting products in its portfolio. Detailing how to manage customer expectations in such a configuration is outside the scope of the present paper,

that deals with contractual issues and marketing issues which are too broad for this discussion.

- to acquire a patent thicket, as a defensive move against other patent holders.
- to acquire personel, as a way to hire lots of individuals simultaneously.
- to acquire control of elements of its supply chain. In the software industry, this could be to take over ownership of a third-party vendor to get control of a specific software component considered strategic, or to get control of a professional services company that would aid in deployment of software products.
- to invest capital, but this particular strategic goal is not our focus here, hence we shall not mention it further.

Acquisitions seem to be difficult to perform successfully: the acquirer needs to ascertain the representations and warranties presented by the target, be sure of the validity of Intellectual Property claims (that is, it should authenticate the pedigree of software source code, judge desireability of any patents), perform its due diligence, as well as be able to retain key personnel (Given the well-known differences in individual productivity in software development, it is only natural that one wants to retain top performers). In addition, acquisitions are sometimes subject to regulatory approval, more so as the size of the entities warrants the attention of governments and their agencies. International acquisitions may even be subject to approval from multiple foreign authorities, which could be governments or others, such as the European Union.

In my professional life, I happen to have been part of a Merger and Acquisition of a small (then about 200 engineers) company by a much larger one (2000+ engineers at the time).

- The smaller company was very nimble, had a very flat organization, and engineers were segregated in a few groups, but each individual could check changes into source control daily. This very much relied on engineers exercising extra caution when pushing a change, to be sure to not break the daily build. This approach worked but did not scale.
- The larger company was using a hierarchical organization, and engineers could not check changes directly into the mainline, the organization was divided in groups, each further divided into teams. The company was using a multi-level delivery system, so changes made by individual engineers would be visible to their team daily, then submitted one more level up weekly. This approach was slower but scaled much better.

Figure 4-2: Example of an acquisition situation.

4.2.2.2 Mergers

In the "M&A", people often forget that the *Merger* part is often the most difficult. Imposing policies from the acquirer onto the target company (although sometimes, surprisingly, it can also be the other way around!). Mergers are often accompanied by a notice that there are great synergies between the two entities and that the new joint entity will somehow be more than the sum of its parts. However, there can be many strategic reasons for management to decide *not* to integrate the two entities:

- the target markets of the two companies are different, or represent different segments, and keeping two separate companies (maybe two separate brands, to avoid diluting the higher-value brand) can be very meaningful.
- the operating procedures are so very different than integration is deemed impossible
- there could be contractual or legal issues preventing integration (we are not talking of regulatory approval of the acquisition here, this is supposed to have been obtained. Instead, we are mentionning the fact that one entity could be operating under the control of some regulatory authority, while the other one cannot).
- the acquiring company manages a portfolio of companies, and the acquisition target becomes one of several in the portfolio. Even when acquiring multiple companies in the same product space, the acquirer can very well have a strategy to own multiple products marketed to different segments. We will not be detailing the strategic management of marketing in this thesis.

For an operations standpoint, each company operates initially in isolation, and a merger between two such entities is going to cause disruption to operations of both of them. Even if one company's operating procedures are chosen to be the gold standard for the new entity (we are not even going to talk about the case where additional transformation is layered on top of an already demanding and difficult transition), the entity that gets modified to fit the other sees a complete rethinking of the way it operates. For the software development operation, this can mean a new source-control system (possibly with loss of data from their old system), a new build system (possibly with changes in naming conventions, updates to copyright notices visible in the end-product), a new defect tracking system (possibly with loss of historical data from pre-acquisition defect history). Even for the entity whose procedures are left untouched, absorbing an acquired entity is very different from organic growth: the merger is an externally-imposed stress on the system. This can cause fluctuations on the operations that disrupt normal operations.

The software development organization is a very large system, which can be seen as a system of systems: external growth is one way more subsystems can be integrated in the overall system. Sometimes it is desirable to keep the new (i.e. the acquired/merged) system separate from the master system: for instance to protect one from changes in the other (this can be meaningful if the two systems are in fact really separate, it is a management choice whether to merge the two to benefit from synergies). Typically in acquisitions, the acquired system has independently development many similar components (such as an OS portability layer), and while if that system had been developed on top or in conjunction with the main system, such duplication would not exist, de-duplicating such components may not yield any measurable value (and indeed performing de-duplication can very well require changes in both systems as assumptions and incompatibilities are uncovered).

Integration is one possible choice but it is by no means the only choice. An organization's leaders may very well decide to leave the new acquisition separate from the rest of the organization. There are many valid strategic reasons for doing so:

- marketing reasons, if the two entities cater to different customer segments.
- avoiding to overload of the two entities: maybe one is going through some significant change, and to avoid making it go through too many changes at once, managers may want to defer merging until a more appropriate time.
- business reasons may dictate the acquisition but not the integration. For instance, one may want to eliminate a competitor from the market by acquiring it, and then EOL its product line, eventually forcing customers off their legacy systems and onto a new system provided by the acquirer. This is one way to acquire a competitor's installed base. If the two systems produced by the two entity have similar features, it may be possible to migrate legacy data from one to the other. Progressively withdrawing support and maintenance is a way to incentivize customers to migrate.

Merging an external entity acts as a step function on the system, we know how such change in inputs can drive the behavior of non-linear systems. Integration concerns all functions of the enterprise, indeed the company will need to harmonize on all aspects:

- HR systems,
- ordering systems,
- marketing, if it is to send clear messages to customers.
- research and development capabilities. These include processes described in section 3.2.

In the context of the software enterprise, it is this last aspect which most concerns us. One special case of this is the acquisition of a partner company, in particular one that develops products on top of the product platform of the company: in this case, merging software production systems is easier. In less favourable circumstances, the acquired company may need to rewrite or *port* its software application to the product platform of the acquiring company.

This concludes our chapter on challenges of external growth. In the modeling sections of chapter 5, we consider all other choices to have been made beforehand, and look at the crucial point in time when two products lines are merged, with management implications for such an event.

4.3 Integrating third parties

Many large software development projects are at some point compelled to reuse third-party components, often called *COTS*, for *Commercial Off-the-Shelf*, components. In this chapter, we will examine the reasons for this to happen, the pain points that the product line endures because of this, and the levers managers can use to make the best of third-party components.

4.3.1 Why use third-party components?

The need to use external components can happen for any of several reasons:

- developing one particular component, even though technically feasible, does not make economical sense.
- the component in question is a commodity and there is no point in spending precious resources in developing something that is otherwise available.
- the component in question is a standardized one, and customers expect the system to use that component and none other. In this case, the choice of component is imposed externally. A example of this is the J2EE platform, which includes a very large set of specifications for various components: J2EE users expect the web servers they program for to explicitly respect the Application Programming Interfaces for those components. Although there is nothing that prevents one to write a new web server program that does not implement the *servlet* API, no one would think of doing this nowadays, as no one would use it: compatibility is of supreme importance.
- the component in question resides outside of the sphere of competence of the organization. For instance, there would ordinarily be little point in a software company developing a proprietary database system from scratch, as there exist perfectly good ones which already dominate the market.
- the product system needs to interface to other systems. As soon as the system becomes part of a larger ecosystem, it will require some kind of interface to other systems. These interfaces can manifest themselves in many ways:
  - in the spirit of *Electronic Data Interchange*, the interface could be done through the interchange of files. The format of those files becomes the de facto interface
  - when the other system is a legacy system, or some kind of *Relational DataBase Management System* (*RDBMS*, for short), these usually provide client-side APIs that other systems can call to interface to the other system

  - some have socket-level interface, i.e. some wire protocol (which could be a TCP level protocol, for instance, or a HTTP request based protocol, as is fashionable in the world of REST services and SOA) that one can write a client for.

This is often called a *make/buy* decision, it can happen for many reasons:

- it could be to benefit from commodity software products. Avoiding to spend the time and effort to re-develop some component that is widely available is a potent influence, as actually re-developing such components could be largely a waste of money. Note that there may very well be very good reasons to choose the re-development path anyway, such as creating a new implementation unencumbered with licensing issues, or making the new implementation integral to the software system developed by the company in order to gain access to features otherwise unavailable.

- free alternatives may be available as open-source or public domain software.

- buying from a partner or vendor.

Any of these may require versioning, if only to support evolution on the other side of the fence. Therefore the system product manager must include the management of those third-party components as part of his whole-product system.

4.3.2 Selection of third-party components

Many factors participate in the selection of third-party components:

- Vendor presence and prestige. For instance, any product that has a RDBMS interface must support Oracle products if the target market is enterprise customers, other vendors may be supported depending on the balance between cost of development and interest from customers.

- Intellectual Property concerns can have a very large impact on the decision. Certification of origin as well as licensing terms count. Open-source software may appear to be "free", but have licensing terms adverse to the actual policies of the company. For instance, some companies avoid at all costs software licensed under the terms of the GNU General Public Library, for fear of contamination. Other popular open-source licenses are better received by the commercial software development community. Legal department often shy away from allowing usage of open-source software for fear of future litigation, especially with regards to authenticating the authors and confirming their actual rights to give away their chosen licensing terms. Similarly, indemnification concerns can put a brake on the reuse of widely available software.

- a balancing factor is the need for the company to avoid spending resources developing a clean-room implementation of a software component that is already commoditized.
- the actual mode of distribution of software is important, software that is shipped to customers sees different effects of the above factors than software published over the world-wide web.
- if the component is integral to the product architecture, in-house re-development may be warranted. If it is peripheral, it may be possible to adapt the product architecture to isolate the component such that changes only have local impact. This is in effect de-coupling, maybe by creating an interface to the third-party component.

Once a component has been selected for inclusion within a product, there can still be conflicting requirements for two products within the product line: two may need different and incompatible versions. The best position to be in would be to certify a single version, but in some cases one must support several and needs to adjust the software architecture to isolate one from the other.

4.3.3 Third parties as partners

Let us note that usage of third-party components within the product system is one possible direction: the company reuses a third-party component. But there is another direction: some third-party develops components which are destined to fill a void in the company's product suite. This is part of the larger theme of product platforms, and how some companies have been extremely successful by creating software platforms where many partners can thrive and increase the utility of the platform. Having the ability to create positive network effects has been well documented in the literature.

4.3.4 Third-party component evolution

We may observe that when procuring a component from a third-party vendor, the software system becomes itself a customer of that vendor, and new released versions are fielded *to* the organization (i.e. in the opposite direction of fielding the organization's product to its customers). This makes the organization susceptible to de-stabilization and vulnerable to changes in the third-party component:

- new defects can occur in the component, in which case integrating a new version causes rework in the organization: maybe in the form of a backing-out of the new version of the external component, or in compensatory action within the organization
- old but until-now innocuous defects can become visible with the new version of the component. This typically happens when the component tightens or otherwise makes

apparent its requirements for being hosted within a product. An example is given in Figure 4-3.

We should note that the release schedule (if there is even one: some products, notably open-source ones, do not even have a fixed release schedule, and produce releases when defect fixing mandates it) of the third-party components usually is different than the one the software development organization uses. The organization must pick which *release train* of the third-party component to build on, and synchronize the import of that released version of the component into its own schedule. [Berczuk 2002, ch. 10] gives details on how this is implemented in Source Control, the reasons for this synchronizing to happen are typically:

- in order to get critical fixes, it may be required to upgrade while in flight. That is, the upgrade might have to happen even though it was not scheduled. This happens typically for security-related bugs.
- in order to upgrade the component to get access to new features, it may be necessary to wait until the next available release train starts. It may not be the next departing train, but the next one with feature content such that the tasks that need accomplishing to operate the upgrade can be included in the plan.

The engineering manager must balanced those aspects to pick the best time to import a new released version of a third-party component. In case the organization itself finds a critical defect in the third-party component, there will be pressure to develop a fix locally: this tendency must be balanced against the long-term needs of the organization. In particular, one must be careful to not end up with a locally modified version of the component that the organization needs to maintain for itself, as this would go against the very benefit of employing the third-party component in the first place. Another aspect of this issue is that the whole componet must be recorded locally in source control within the organization, its build must be automated and should at all costs avoid automated download of further dependencies, as builds *must* be idempotent.

Another point of interest is that third-party components are often systems onto themselves and thus may include directly or depend on other third-party components, and therefore it may be important to perform all upgrades at the same time, as the company does not want to spend resources on testing combinations of components that have not been tested and approved independently. This goes to the same point as previously: the organization should be careful to not have to perform quality assurance functions in place of the provider of the component. In conclusion, this goes to the same point as previously: third-party components must be carefully selected based on quality criteria such that the long-term benefits outweigh any resource spending within the company on the use and integration of the components.

Sometimes the organization will select a third-party component and then be forced to take over maintenance of the software itself. This happens when the third-party vendor loses interest in, or deprecates the component, or simply goes out of business. In all these cases, the organization ought to be careful to select only components it can get in source form (*white-box* components, as opposed to *black-box* components): even though one may elect to integrate the component into the product build system as binary components, it is crucial to preserve the ability to fix the software, or port it to a new environment (See Figure 4-4).

Third-party components often unintentionally breach modularity and induce bizarre interactions in large systems. In one system I worked on, upgrading the Java Development Kit used induced an interaction with our software: prior versions of the JDK were compiled on the target platform with an earlier version of the C/C++ compiler and linked with the corresponding version of the C runtime library, when we upgraded it, we found out that loading the Java Virtual Machine into our process changed the TTY mode of the standard input! We had been immune to that interaction as long as we were not building our product with the same C runtime (it was in fact loading two copies of the C runtime shared library into the running process). This is another example of why systems engineer want to enforce component isolation to ensure that components can indeed function independently.

Figure 4-3: Third-party components and modularity.

One third-party component I worked with presents an interesting case. This component was a portability layer on several platforms. The vendor went out of business in the 1990s, yet the company that had incorporated it into its own product line outlived the vendor. Almost twenty years after the vendor going out of business, the company had to start supporting 64 bit operating systems, as a result, it was compelled to either get rid of the system, or perform itself the work to port this component to 64 bit environments. The latter strategy was selected by management, and was conducted successfully. This was possible because the company had had the foresight to acquire a source code license to this third-party component, a risk-mitigation strategy which turned out to be crucially important years after the initial choice and sourcing of this component. As a result, the software system in question is still available commercial on platforms the original designers of the components never thought of.

Figure 4-4: Example of risk with third-party component.

4.4 Software Product Lines

Large software development companies typically sell multiple products grouped in families; if those families are entirely independent within the organization, then we would consider them to be separate. Products are also going through multiple releases, their development is ongoing, possibly going on indefinitely; new members join the product family, some member leave it, while others are developed continuously. This is exemplified by Figure 4-5, where Product 1 starts at release N-2 and stops at release N+2, while Product 2 sees ongoing development into the future, and Product P starts at release N. Product 1, 2... P form the product family, these could, for example, be Microsoft®'s Word® , Excel® and Powerpoint®. In the time dimension, releases N-1, N, N+1 could be the Office 2003, Office 2007 and Office 2010 versions released by Microsoft®.

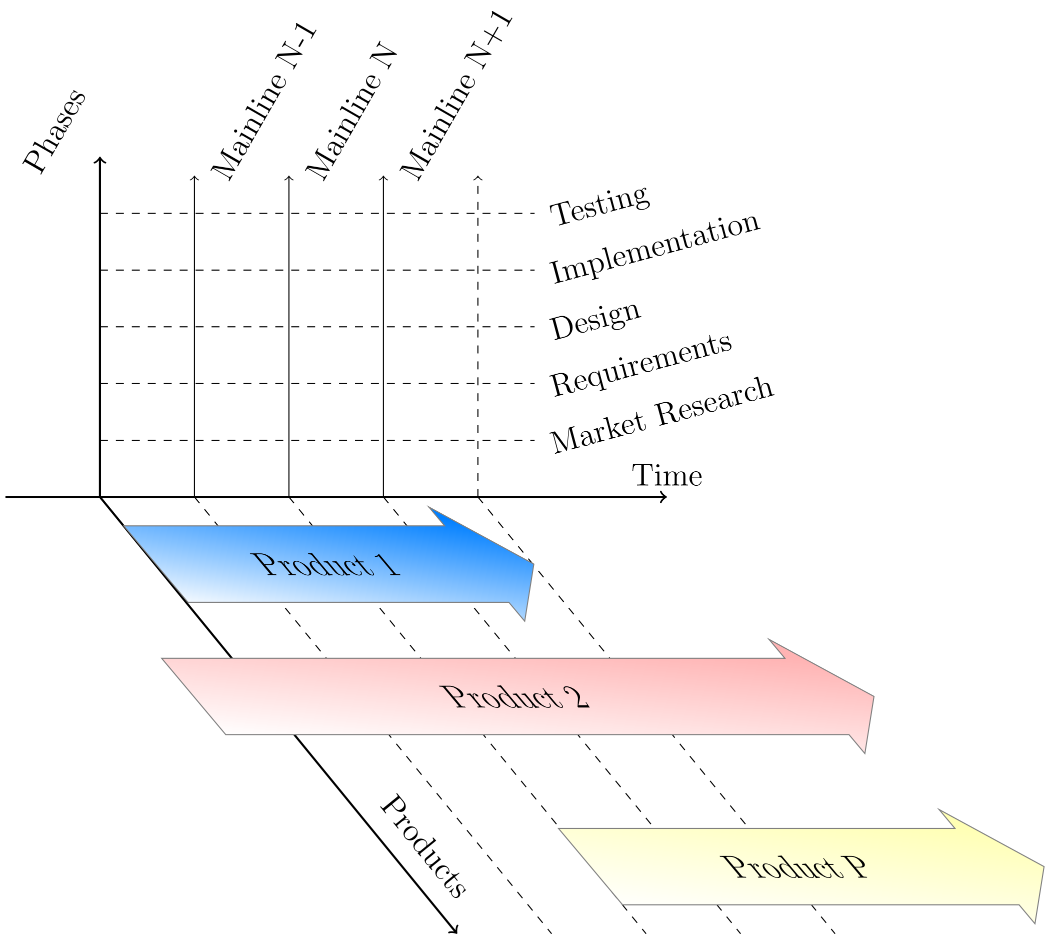


Figure 4-5: Software Engineering along three dimensions.

Products or families of products that are developed jointly (i.e. sharing source level components, merely sharing processes and scheduling, or all of those plus some level of coupling) form a *software product line*. The benefits of product line engineering have been documented in the literature [Bosch 2000, p. 288]. Products within a product line are developed simultaneously, in lockstep, or otherwise share scheduling constraints.

Software product lines contribute one additional layer of complexity in the management of large software development efforts. Large software companies typically maintain multiple active branches, sometimes these branches are very long-lived due to regulatory constraints.

How to manage a portfolio of applications is outside the scope of the present thesis work, instead we assume that the applications in the portfolio have been selected and that it is fixed. Product and feature selection are here assumed to have already taken place, as those topics are outside the scope of the present work. In other words, we shall not examine any front-end processes (i.e. product selection, market segmentation) and we shall ignore the fuzzy front-end, instead we only deal with already decided New Product Developments, which we assume to be known and stable.

For one particular set of products and features, we also assume that the product line we are talking about it already successful enough to sustain itself over multiple successive generations of development work. In particular, we do not talk about initial product or product line introduction (although we will consider introduction of a new product within the context of an existing product line), as this presents additional modeling challenges (for instance, initially no worker on the project has experience with the product, since the product does not exist yet), these are examined extensively in [Bosch 2000, p. 166 and p. 316]. Similarly, in this context, we assume that requirements do not change (i.e. that requirements were completely elicited and are perfectly known: this is an idealized situation, as in reality requirements always change), and changes that will occur will be handled as engineering change orders and scheduled for fixing just like any other defect. This is true both of requirements coming from customers' use cases and of requirements coming from variability changes required by the product engineering teams. The ideal requirements hypothesis is an approximation of reality, as it sees the software organization as a closed system as noted in [Weinberg 1991, p. 159].

To quote Rechtin [Rechtin 1990],

You can't avoid redesign, it's a natural part of design.

When the same team is tasked both with developing the "currently active" branch and with maintaining older product lines (maybe for good cause: it happens often that it is the same engineers that have the knowledge of product architecture and/or details of the implementation), then those multiple branches compete for the same resources, namely the set of capable software engineers. This is related to the *firefighting* issues documented by Repenning [Repenning 2001].

Note that in this work, we call *current* or *currently active* or *active* the one branch which is undergoing active feature development: this is the highest-numbered branch and is where all feature development happens. This is a simplification of reality, because sometimes feature development may be requested by customers on other branches, but typically in industry it is minor work, and is handled much like a defect.

In other words, this is an example of the *Robbing Peter to Pay Paul* archetype documented by [Novak & Levine 2010]:

- one release may be given more attention at the expense of another. For instance, if the *fielded* releases exhibit a highly problematic fault and must be fixed immediately, the manager may elect to divert one engineer from working on the *current* release to work on discovering the nature of the defect and proposing a correction.

- one product may be given more attention at the expense of another. For instance, the Product 1 engineering team might discover an intolerable defect in a shared component, requiring diversion of an engineer to work on said defect, stealing that person away from the work he was scheduled to accomplish in the current release.

The devastating effects of the various active mainlines competing for shared resources, in this case the time of engineering staff, will be demonstrated in **Chapter 5**. This is a well-known problem in the *Critical Path* Method of project management: if every step uses its own safety buffer, then projected completion takes much longer. The proper way is to have a per-project safety factor and to monitor use of the safety. In other words, a holistic view gives much better results. However we should note that CPM applied to several projects competing for shared resources is not exactly equivalent to what happens in product line engineering, which sees additional effects of propagation of rework across active mainlines.

[Pohl, Böckle, & van der Linden 2010] say that

> *Whenever an artefact from the platform is changed, e.g. for the purpose of error correction, the changes can be propagated to all products in which the artefact is being used. This may be exploited to reduce maintenance effort.*

This is true but misses the point that fixes may need adaptation on multiple branches (i.e. may not be identical across all branches). Our model of the system will thus have to account for this additional cost of propagation, due to administrative overhead of merging a change on a branch and to the added difficulty of having to adapt changes to each such branch.

As noted in [Pohl, Böckle, & van der Linden 2010, p. 18] :

> *The stability of the domain is also an important factor for the successful introduction of software product line engineering. If everything changes every half-year in an unpredictable way, the investment costs never pay off. This situation is similar to not understanding the domain well: variability is added that is not needed and the variability that is actually required is not available.*

Without loss of generality, we can consider that re-architecting processes (such as splitting a component in two, adding or removing dependencies, retrofitting commonalities) are activities that are conducted only at the beginning of a release cycle. Doing otherwise adds unnecessary complications: for instance, adding a dependency after a released product has been shipped to customers necessitates additional support in the delivery system to be able to ship such components to customers. A very sophisticated delivery system would be able to support this. However re-architecting and refactoring add no value for customers, these are non-functional features that customers usually do not care about (unless they cause some issue visible to customers, such as system instability or lack of scalability): the enterprise can schedule these changes as it pleases and usually as is convenient with regards to feasibility. This happens when product engineering units need to extend some core component rapidly to meet schedule, and the product-specific extensions, if found profitable to other units, are refactored into the core component in the next release of that component [Bosch 2000, p. 312].

The option to push back features, or schedule them for later, simply is not available in projects, but is possible in product line engineering. In fact, some value to some subset of customers is realized even if not all initially scheduled features actually make it into a given release. Scheduling features for future development and delivery also has advantages to the organization other than workload smoothing and planning: for example, it may present value in terms of time to market, so that the organization gets one iteration of the product to the market as soon as possible in order to capture market share, and it may as well as present value as a marketing device in terms of planning the roll-out of features to the marketplace. Another selling point for customers is that the product line typically provides for ascendent compatibility between successive releases of the product family, therefore even if one feature is not present in the current release, the company can provide a roadmap to customers to entice them into purchasing the current release, in hopes of future benefits from features planned in later releases.

If several re-scheduling initiatives are ongoing simultaneously, they can be ordered and scheduled for delivery on staggered releases: these are then handled as an additional stock of work-to-do, with no visible added-value feature content for customers.

In my experience, the best way to track architecting tasks, which present no user-visible benefit, is to create a defect in the defect tracking system. This is not made visible to customers, as it has no value to them, but is handled inside the company like any other defect. The main issue with not recognizing refactoring and re-architecting tasks formally in the task allocation system of the company is that doing so ensures that these tasks remain hidden from the execution system, which means it is just another form of undiscovered rework. That rework may be known to a few individuals, but short of having visibility in the formal development system, it will never get allocated any resources. This is sometimes known as *technical debt*, which is rework discovered but deferred.

Such a policy means for instance that fixing the defect can be deferred until a convenient time (with regards to the multi-release scheduling mentioned above), and it can be prioritized against other issues.

One of the hypotheses in the present paper is that having multiple active mainlines is costly for the enterprise. Usually there is a fixed cost per active mainline, related to the administrative overhead of keeping the branch data in source control, staffing the build operations for the branch, tracking propagation of changes across branches (forward to correct defects in latest versions of the product, backwards to give out critical fixes to customers using older branches in production). We desire to show that minimizing the number of active mainlines increases throughput of the software delivery system, or conversely that having too many active mainlines decreases the throughput of the system. This notion is easily entertained intuitively as more mainlines cause more opportunities for firefighting across mainlines.

With regards to organization, one should note that having rhythm in the software production process makes deadline clear to everyone at every level in every function: there is no need to communicate specifically about milestones and deadlines when everyone knows about them. Rhythm allows individuals to internalize what the expectations are with respect to schedule, it also gives direction to each individual as to what is the right thing to do at what time: for instance, individual engineers know instinctively that making large impacting changes around the end of a release cycle is not a good thing, since there will not be enough time to properly test the new version of the product with those changes. Refactoring tasks and other architectural changes are thus only performed at the beginning of a cycle so that the QA team have time to perform adequate testing and build the proper level of confidence in the product. Similarly, other functions in the organizations all align on the same schedule: financing, budgeting, hiring, marketing, etc... all know the tempo and perform in accordance with the deadline [Dikel, Kane & Wilson 2001, p. 237]. In this fashion, rhythm is a very powerful tool leaders can leverage to make coordination *implicit*, to make schedule part of the culture of the organization.

My industry experience has been with an evolutionary delivery of several software systems in a common source control system, using the feature superset approach [Bosch 2000, p. 333]. Having a single repository makes it possible and even necessary to synchronize the release schedules of all the software product lines involved; since those systems are often used together as part of a large IT system, it is important that all integration issues be resolved as early as possible.

Development gets further complicated by issues not otherwise detailed in this document. For instance, upwards and sometimes also backwards compatibility of file formats and wire protocols adds another layer of testing to perform before release : if you have systems A and B released together, the current release of A must work not only with the current release of B, which was developed simultaneously, it also must function properly with prior releases of B, so the amount of testing is roughly quadratic in the number of systems mainlines you integrate together. This slows down the quality assurance work tremendously, as the quantity of work increases quickly. Usually this means that the testing effort is scaled back, manual cursory smoke testing is performed, but not all possible combinations are tested. Preserving compatibility in all possible cases requires an enormous amount of care, and proposed changes that risk it are usually peer-reviewed closely before approval. This is particularly true of the core of those systems, which therefore tend to have a slower evolution rate than the peripheral parts.

There are essentially two ways to manage product lines:

- either maintain the shared assets in a single repository, and instanciate those (i.e. copy) them over to each product repository, then build those independently of each other. One must then assemble the system and perform integration testing. Note that in this case, shared assets may be compiled and configured independently, with the possibly that two systems use two different versions of a shared asset.

- or maintain all assets together in a single repository, have release engineering perform a single master build of the system or systems, then select for each system the subset of all artefacts that compose it. This guarantees that shared assets are strictly identical, as there is only a single canonical version of those.

The first possibility builds cloning/branching of shared assets by product engineering teams, which gives them opportunity to make local adaptations, which make complete system builds difficult, or sometimes even impossible.

A company using the second possibility displays maturity because variability is necessarily bound at run-time to enable this level of sharing of assets. This is called the feature superset approach [Bosch 2000, p. 333] to product derivation. This promotes a software product system that always is in a useable state, and it forces engineer to resolve immediately any conflicts as no two components can require two different versions of the same shared asset: since there is in fact a canonical version, that is what every product engineering team must use.

Finally, we should observe that organizing is about dividing labor, by predicting how much labor there will be and what a good way to divide it is; we shall examine later in chapter 5 how work items actually flow within the structure established by the organization, that is the dynamic, rather than static, view of how work gets accomplished.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

System Dynamics Modeling

Lehman noted in [Lehman 1998, p. 42],

> *The software process constitutes a multilevel, multiloop feedback system and must be treated as such if major progress in its planning, control, and improvement is to be achieved.*

This observation hints that System Dynamics modeling could prove useful as a way for us to formalize our understanding, however approximate, of what is happening in the software production system. This is precisely what we intend to do in this chapter, starting with a short introduction to System Dynamics.

5.1 Introduction to System Dynamics

System Dynamics is a field of study founded in 1950s at the Massachusetts Institute of Technology by Professor Jay W. Forrester, with the objective of providing tools and methods to better understand the dynamics of complex systems. Forrester was able to explain trends in managerial situations in industry by applying concepts from mathematics and engineering (related to control theory) to those situations. He demonstrated that the structure of systems is sufficient to explain non-linearities which are non intuitive and difficult for humans to grasp.

System Dynamics uses the concepts of stocks, flows and feedback loops to model systems as directed graphs:

- a *stock* is a variable of the system, such as a number of people, or a number of tasks. *Stocks* describe the state of the system: they can accumulate, or increase in value; they can also deplete, or decrease in value.
- a *flow* is a rate at which a *stock* changes, similar to a speed measurement.

- a *causal loop* describe a link between elements. The relationship between two elements is a *loop* when each element (stocks or flows) has a causal effect on the other; the effect can be positive or negative. These represent the structure of the system: the philosophy of System Dynamics is that these influence how the system behaves dynamically.

One example familiar to product development engineers would be a simple model of development work with:

- a *stock* of tasks to be performed
- a *flow* of incoming tasks per week, coming from the customers. This rate could be steady (one per week constantly, for instance), or varying (for example, every ten weeks, an increase of a hundred tasks that tenth week). This flow contributes to the accumulation of tasks in the work backlog of the developer.
- a *flow* of outgoing tasks per week, modeling the fact that the worker performs some tasks, thereby depleting the number of tasks remaining.

The interested reader can learn more about System Dynamics modeling from the literature, in particular [Sterman 2000].

5.2 Applicability of System Dynamics to Project Development

Early work in this field, such as [Abdel-Hamid & Madnick 1991], recognized the usefulness of systems thinking and system dynamics modeling to help with this understanding. [Acuña, Juristo, Moreno & Mon 2005] surveys software process modeling, citing the integrative model from [Abdel-Hamid & Madnick 1991] for its pioneering work.

The System Dynamics view of the world is that, although many workers and managers blame external factors for the failure to complete their projects, many performance problems come from the fundamental dynamics of projects, and thus understanding these can help project leaders manage those profitably. Understanding the dynamics and what lever to employ to counter undesirable dynamics is therefore very valuable to managers, or put another way: the goal is not Taylorism but comprehension as a way to help avoid policy resistance.

Note that as Novak and Levine write in [Novak & Levine 2010], *a system dynamics simulation can provide useful qualitative conclusions about the general behavior of the causal loop structure*: in other words, SD models are not meant to provide precise quantitative evaluations of behavior, but rather give an idea of what the general trends are in a system. Here we assume that people are interchangeable, which they are not, as seen in Section 2.10: we do not deal with the additional texture that different persons behave differently, have various capabilities, varying levels of ability, various levels of experience on the project.

We consider here that these difference will be averaged over the long running time of the simulation.

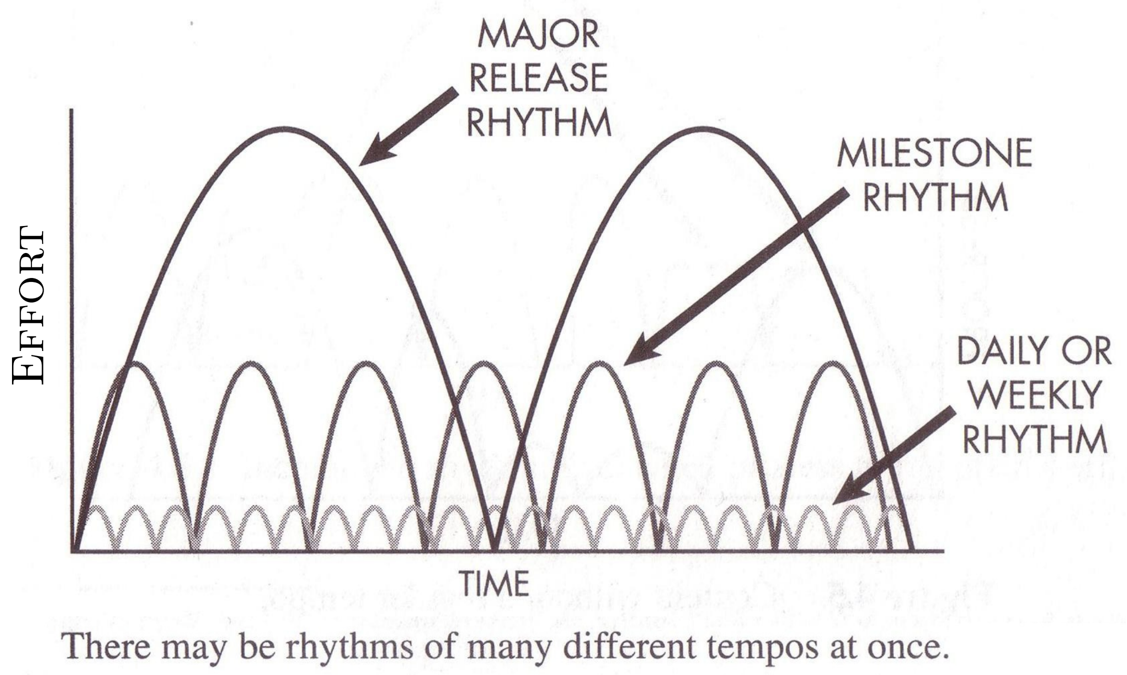


Figure 5-1: Internal rhythm of a release, from [Dikel, Kane & Wilson 2001, p. 75], only considering feature development.

The software development literature makes it clear that development projects have an internal rhythm: in particular, [Dikel, Kane & Wilson 2001, p. 75] shows the Figure 5-1 reproduced here, where the effort expanded by the workers and the cumulative expanded by the organization varies in relationship to the schedule. This view is representative of reality of projects, we shall see in later sections that in the case of parallel engineering of multiple releases of the same product family, the rhythm happens simultaneously on multiple releases.

Here we are careful to consider models as an approximation of reality, yet we contend that such an approximation can yield insights which are important to the manager. All the dynamics that can be found in actual project performance can be explained by accumulation processes and feedback processes, therefore the next few sections will look at those, with a slant towards large-scale software engineering.

5.3 The Rework Cycle

The Rework Cycle is a well-known archetypical feedback loop in System Dynamics. Figure 5-2 shows a causal loop diagram of this phenomenon.

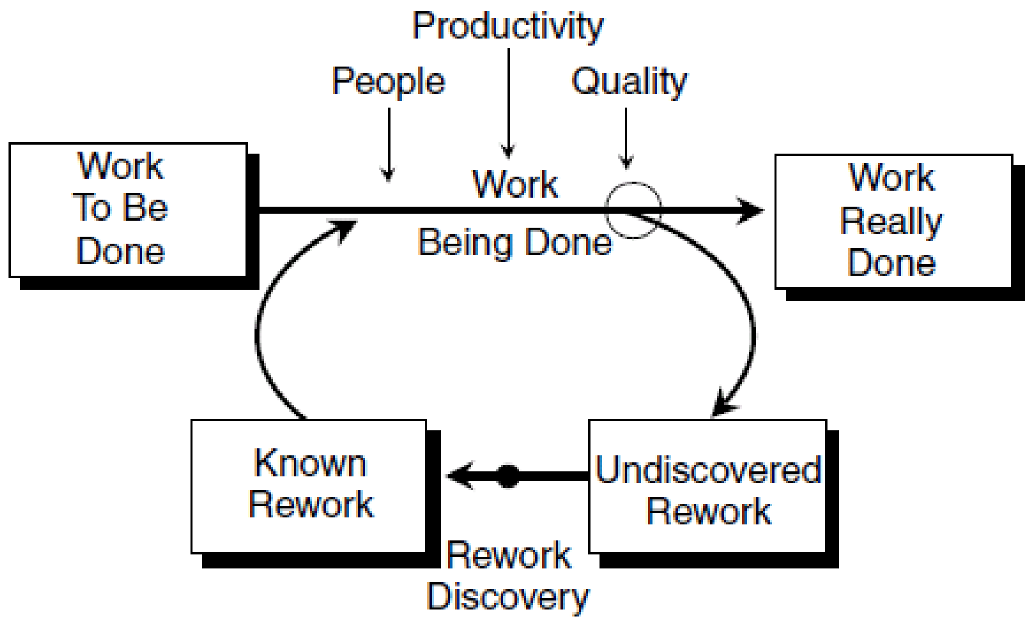


Figure 5-2: Rework as a causal loop diagram, from [Lyneis, Cooper and Elsa 2001].

The stock of *Work To Do* accumulates tasks which need to be performed: this is initially set to represent the tasks that the manager has decided will need to be performed to achieve his goals, which could be for instance to implement a given set of features, or to add some improvements into his product. *Work To Do* is depleted by the collective action of engineers accomplishing tasks. When performing tasks, engineers accomplish some fraction of them perfectly, such that the work product of those is correct and will never need to be revisited. But they also accomplish a fraction of them incorrectly, which could be:

– because they did not perform the task in the right way, made some mistake,
– or because they perform the task exactly as they should have, but they performed the *wrong* task: this could have been caused by incomplete or unclear requirements.

Regardless of the reason, in this case, the work product is defective and necessitates subsequent rework. In light of this, while there is initially a stock of *Work To Do*, the output of this system is really two stocks: *Work Really Done* and *Undiscovered Rework*, which feeds backs to *Work To Do* as it is discovered. This discovery of rework lags as engineers work to develop the product, but others discover issues with a lag. This feedback loop models one core issue many projects have where the work product of some task is not known to be defective until much later. In Figure 5-2, the rate of doing work is clearly related to the productivity of the workers performing the work, while *quality* is related to the relative proportions of *Work Really Done* (called *Work Done* in the rest of this document) and *Undiscovered Rework* in the number of tasks processed by the workers.

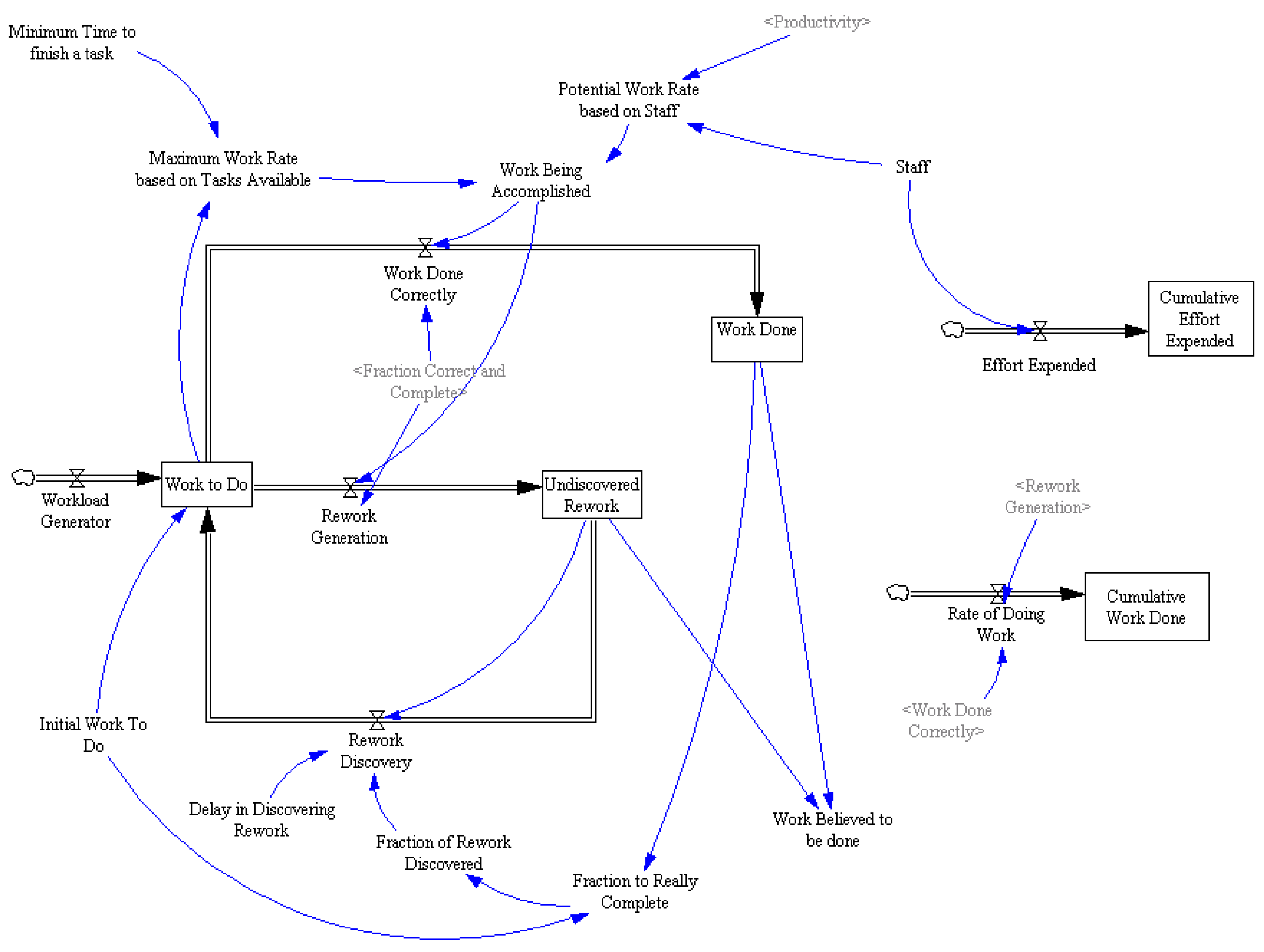


Figure 5-3: Rework loop modeled in VENSIM [Lyneis 2012].

In System Dynamics modeling, it is typical to separate, as on Figure 5-3, the notion of *Work Perceived Done* from that of *Work Really Done*: *Work Perceived Done*, also called *Work Believed to Be Done*, is what many engineers and managers base their schedule estimates on, while accumulation of *Work Really Done* (denoted *Work Done* in the figures presented here) and depletion of *Work To Do* are the two quantities which really matter for project management.

Undiscovered defects are typically a dramatically bad thing to have. The reader should compare Figure 5-4 and Figure 5-5 to see how having a *Fraction Correct and Complete* less than 1 impacts the rate of depletion of the *Work To Do* stock. *Fraction Correct and Complete* is the proportion of work performed that goes into the stock of *Undiscovered Rework*: a FCC of 1 means all work is correct and complete, a FCC less than 1 means that some work is directly correct and complete, but some fraction is initially incorrect or incomplete, and thus becomes *Undiscovered Rework*.

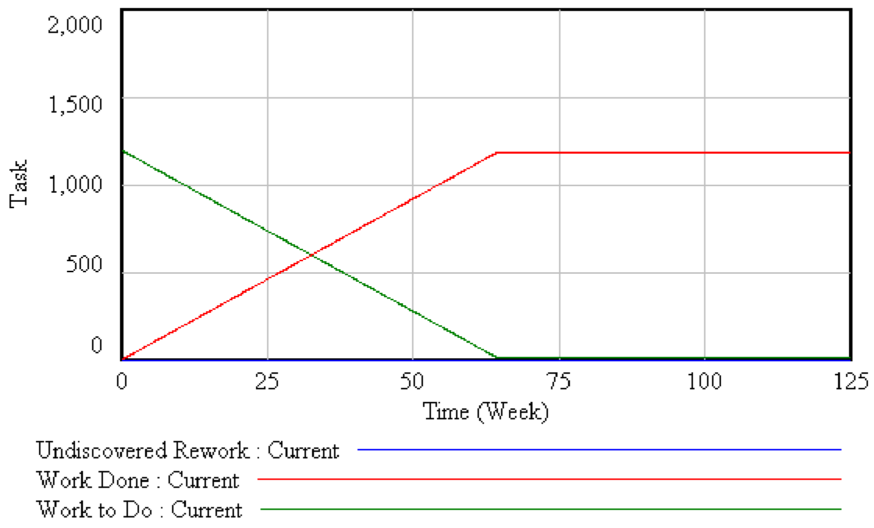


Figure 5-4: Perfect Fraction Correct and Complete

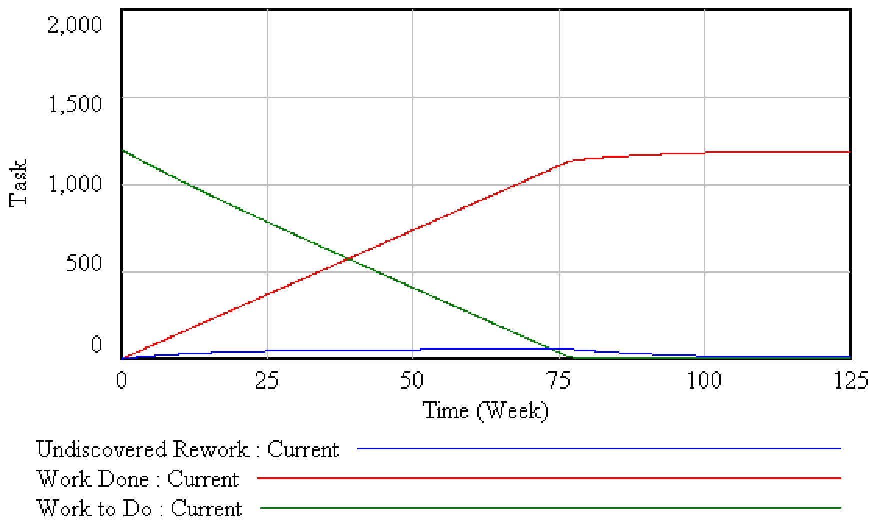


Figure 5-5: Fraction Correct and Complete of 0.8

Engineering leaders should recognize that errors will happen and thus rework will happen, and that cannot be avoided. Nevertheless, leaders can and should put in place policies that decrease errors and keep undiscovered rework manageable (even though undiscovered rework is a quantity unknown and impossible to measure, factors that affect it are known). System Dynamics modelers are always careful to distinguish between "actual" and "perceived" value of variables. *Work accomplished* is a typical example, it is divided between work actually done (i.e. that will not require rework) and work that seems done but needs revising (i.e. undiscovered rework).

5.4 The Rework Cycle in Software Development

In software development, one way the rework cycle manifests itself is the situation where one software engineer creates some new source code, writes and runs unit tests for it and then makes it available for testing to each of the successive consistuencies shown in Figure 3-6, and the person doing the testing (either in-house quality assurance engineering, or possibly in-the-field customer usage) discovers a defect.

Because the feedback loop to discover the defect and start the corrective work can be long, the project as a whole builds on top of defective work products, generating unnecessary and incorrect work, requiring even more rework to finally get an acceptable revision of the work product. Such a feedback loop explains why the cost of fixing defect rises with the moment of defect discovery, as seen in Figure 2-8.

[Rahmandad & Hu 2010] documented many of the possible formulations of the rework phenomenon. In the present work, we shall use the simplest formulation rework cycle, as we should expect the qualitative results to be similar enough for our purposes, leaving use of other formulations as a topic of future study. Figure 5-5 shows an example of how rework affects completion of work.

There are two kinds of defects:

- defective work products, whose defects can go undetected for some time. This may be because under some circumstances, the work product functions properly and does not appear defective, some special circumstance is required to make the defect apparent.
- defective work products that immediately stall the software delivery system. For instance, any error that breaks the build (i.e. that is detected by the compiler), or any error that causes visible regression in the unit test suite replay. These can (and indeed must, as they prevent others from carrying on their work!) be corrected as soon as possible. In effect, rework of this kind admits no delay in being fixed (as opposed to "normal" rework, cf [Abdel-Hamid & Madnick 1991, p. 75]).

In addition, once a defect has been detected, and rework initiated to correct it, submission of software source code changes to fix the defect is itself work that is susceptible to being defective in some fashion. This is called the FIXES THAT FAIL pattern, shown on Figure 5-6: the fix itself possibly has some unintended consequences that constitute feedback in the system, and contribute to more work to do in the future. An example of this is depicted on Figure 3-8 in Section 3.2.2.

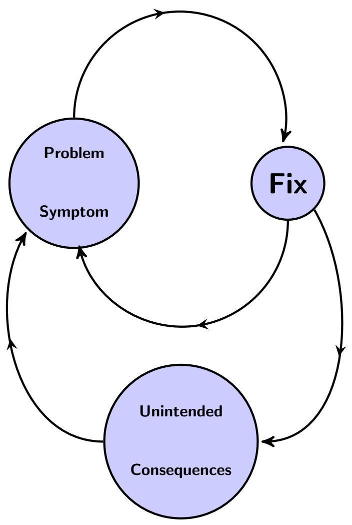


Figure 5-6: Fixes that fail a.k.a the rework loop, archetype from [Novak & Levine 2010]

Ideally, all work products are initially defect-free and there would be no rework ever required, this corresponds to a *Fraction Correct and Complete* of 1 in the SD model on Figure 5-3. In reality, that is hardly ever the case, and downstream tasks usually discover some number of defects, causing rework.

[Reichelt & Lyneis 1999] make the point that

> *A rework cycle and its associated productivity and quality effects form a building block. Building blocks can be used to represent an entire project, or replicated to represent different phases of a project, in which case multiple rework cycles in parallel and series might be included. At its most aggregate level, such building blocks might represent design, build and test. Alternatively, building blocks might separately represent different stages (e.g., conceptual vs. detail) and or design functions (structural, electrical, power, etc...). Similarly for build. In software, building blocks might represent specifications, detailed design, code and unit test, integration, and test.*

Following this, we may want to think how this notion of *building block* can be applied to the *Rework cycle*:

- to represent phases of a project, which we will call *stages* in Section 5.5,
- to represent design functions in a project, as will be seen in Section 5.9.

This concludes this short introduction to the rework cycle, we will devote the next sections to using the rework cycle in various ways to represent what happens in software product line engineering.

5.5 The Rework Cycle in presence of multiple stages

In this section, we want to examine the effect of rework when the work processes are divided into multiple stages within a single release timeframe. This is typically the case as design work is separated from implementation work, which in turn is separated from Quality Assurance work, which may itself present multiple stages. This is depicted on the following Figure 5-7.

This phenomenon is worsened when delays are added. For instance, if the quality assurance steps can only be performed with a delay: this may be because the daily build is broken, or because there is not a *minimum testable product* yet, as defined in Section 2.6.

For physical products, going from concept to physical realization, there are often multiple successive stages, each with feedback to one or more of the previous stages. These stages are often characterized as “engineering” vs “manufacturing”: this does not apply literally to software product as there is in fact no physical realization. However even for pure software products, there are two stages that occur after the “coding” stage that could be considered as “manufacturing”:

- the build phase, in particular if the system is so large that building (i.e. compiling, linking, etc...) takes non-negligible time. In my experience, systems whose size range in the million to tens-of-millions of lines of code usually take non-trivial time to compile and link.
- the test phase, where QA engineers install the product like a customer would and run manual scenarios to validate feature content. There could even be multiple stages of testing (unit testing, system testing, then reliability testing, then scalability testing, then other kinds of PCS, which is *Performance, Capacity & Scalability*, testing).

One example of modeling multiple successive stages is shown on Figure 5-7: the blue arrows represent the links between stages, the forward arrow models the fact that one stage pushes work onto the next, and the backward arrow represents the fact that some issues are discovered in the second stage but can only be fixed in the first, and so are pushed back to the initial stage for rework.

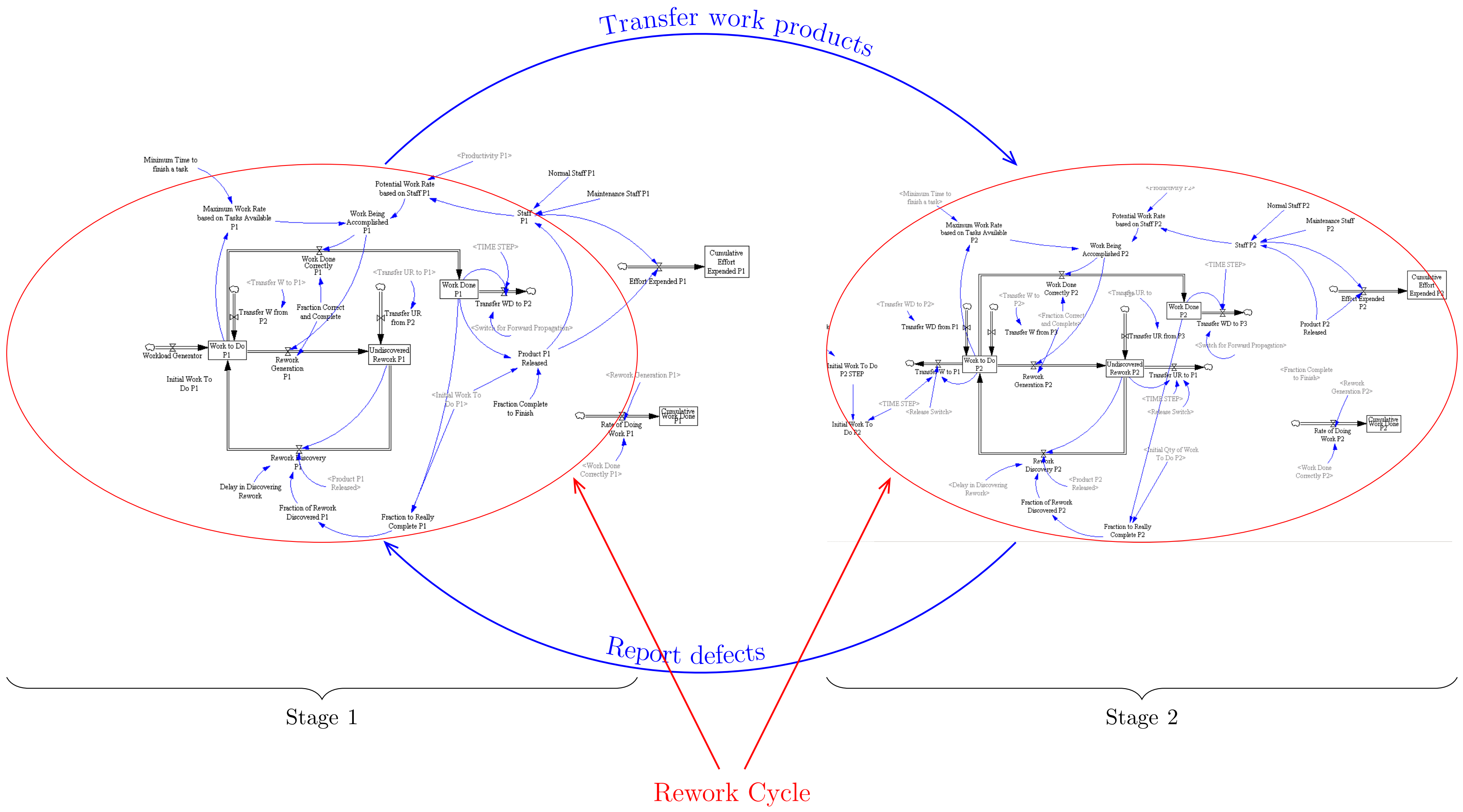


Figure 5-7: Example with two stages, adapted from ESD.36, courtesy of Professor James Lyneis [Lyneis 2012]

5.6 The Rework Cycle in presence of multiple branches

The software product development organization will typically have multiple branches (Note that we use the two terms *branch* and *mainline* interchangeably). Unlike a project, what we have here are N releases operating in parallel all the time, and periodically a new mainline is created, and an old mainline is discontinued. Choosing exactly how many active branches are allowed is a management choice; Weinberg states that large organizations usually converge to having two releases per year [Weinberg 1991]. In this research, we use a bi-annual rhythm as an example, but expect that readers would adapt the model to their own circumstances. Many other rhythms are possible, or even desirable, and the choice of rhythm may be externally imposed, such as a yearly Christmas release schedule in some industries (such as the consumer game/entertainment industry).

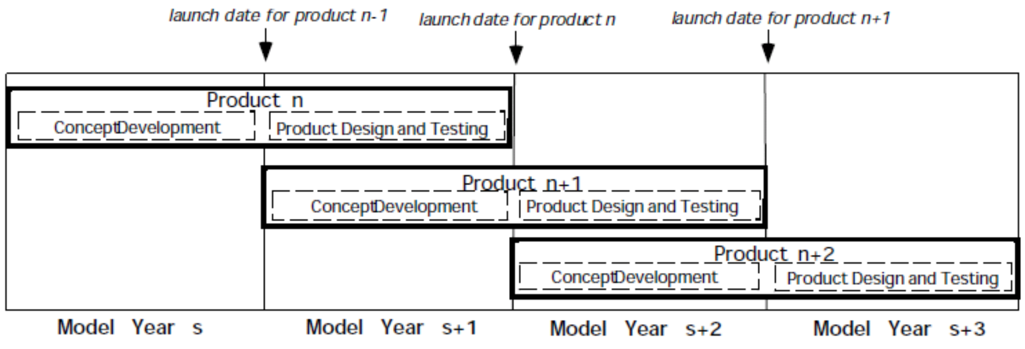

launch date for product n-1
launch date for product n
launch date for product n+1
Product n
Concept Development
Product Design and Testing
Product n+1
Concept Development
Product Design and Testing
Product n+2
Concept Development
Product Design and Testing
Model Year s
Model Year s+1
Model Year s+2
Model Year s+3

Figure 5-8: Repenning's model-years, from [Repenning 2001].

The effects of sharing engineering resources between multiple active branches have been extensively studied by Repenning [Repenning 2001], who calls them "model years", using terminology from the automotive industry. Repenning's work studied a simpler version with exactly two stages called *Concept Development* and *Product Design and Testing*, and considered that once the Testing stage is finished the work on that specific model-year is completely done. The situation we want to study here is different, with three stages, including an additional stage that feeds work into the system perpetually, as explained later in Section 5.7.

This situation is not simply multiple rework cycles happening independently on multiples branches, as it is in fact the very same set of engineers working on all of them, switching from one to the other as emergencies or scheduling dictate. This relates to the *Critical Path* method of project management, invented in the 1950s, which models the project as a graph. The CP graph is a set of related activities, where each node has edges to the nodes it depends on; this dependency graph can then be used to compute the longest path of activities in the project, which is the *Critical Path.* This method works well when the resources are infinite; in realistic projects, resources are finite and shared, consequently the *Critical Chain* method was created by [Goldratt 1997] to account for those conditions. However resources in the *Critical Chain* method are typically machinery or inventory, whereas in the present thesis the most valuable resource of the project manager is the staff workers.

Branches that have been released to customers switch to maintenance mode: this means that no additional feature work is ever performed on those branches, and all new capabilities are developed on the in-development (also known as *current*) branch. New capabilities are only considered for the current and future branches, management typically limiting the amount of work going into the current branch, and prioritizes new capabilities to spread their development over multiple future branches.

In addition, discovery of rework on one branch typically induces rework on other branches:

- if a defect is discovered on one of the newer branches, the manager must decide whether to add to the task queue of earlier branches the tasks that would need to be carried out to fix the defect there. This process is called *backporting.*
  - Sometimes the defect will be too costly to backport, and no change will be made to earlier releases. This happens when deployment costs are too high, or too few customers are impacted by the change (as could happen if in fact no customer using those released products ever encountered the defect).
  - Sometimes backporting is simply a matter of copying (this process is called *merging* or *importing*) the changes to the older branch, so the rework task is not nearly as costly as those on the original branch the defect was discovered on. But even in such a favorable case there is still testing, distributing and deploying costs that need to be considered.
  - In the worst case, backporting is not possible: this happens when the codebase has evolved significantly between branches, and then it is not just a matter of copying the fix: one needs to engineer a new and different fix.
- if a defect is discovered on an older branch, then it will typically need to be *ported* forward to all later branches. There are exceptions to this situation, for instance when the defect is in an area of the codebase that has been deprecated or removed.

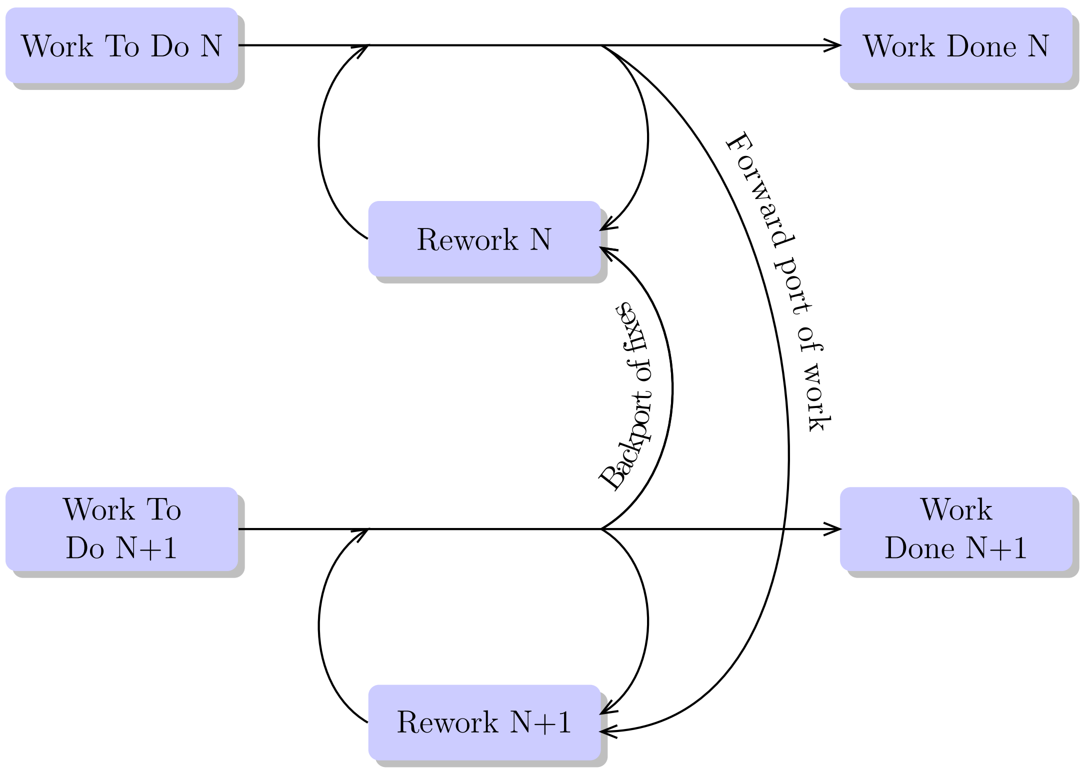


Figure 5-9: Example with two parallel branches, named N and N+1, each feeding rework into the other. Note that cross-branch rework is subject to some damping factor, as work is less demanding the second time it is performed.

In summary and as shown on Figure 5-9, while there is rework discovery happening locally within each branch, this system appears to be working under a grander rework cycle across all active releases. This leads us to classify defects in several classes:

- defects that should have been caught when the associated capability was first introduced in the software product line. Issues of this kind really are defects according to the original requirement (i.e. the work product does not match the original specification).
- defects due to implicit or new requirements that could not possibly have been engineered for, or even known, at the time the associated capability was first introduced. Issues of this kind are retrospective requirements (for instance, for a web product to support a future version of a web browser that has not even been released at the time of specification), typically they concern maintenance tasks related to keeping the product up-to-date with regards to changes in the product environment (browser changes, compiler changes, operating system upgrades, new standards). Weinberg says that *in software, conformance to requirements is not enough to define quality, because requirements cannot be as certain as in a manufacturing operation* in [Weinberg 1991, p. 31], and indeed requirements are not fully known or understood at the outset, and so they

do change or are refined over time. These changed requirements usually appear on a new branch as a new requirement, even though they really are about fitting existing code to changes in system interfaces.

This is an occurrence of the *accidental adversaries* archetype documented in the archetype catalog of [Novak & Levine 2010, p. 13] , where *two parties destroy their relationship through escalating retaliations for perceived injuries.* In this archetype, A and B execute concurrently, but the actions each takes towards their own goal undermines the progress of the whole system. In the context of software mainlines competing for staff, each mainline optimizing locally does so at the cost of lesser progress for the system as a whole: one mainline retaining staff instead of sharing it, or taking shortcuts in the work performed, hurt the overall system.

Repenning only considered the case where at most two branches are active at the same time: one in the *Product Design and Testing* phase (this is model-year S) and another in the *Concept Development* phase (this is model-year $S+1$) in Figure 5-8.

Each time a new timed-boxed release is started, a new branch is set up. This new mainline already has some stock of work allocated to it, which contains:

- new feature development tasks,
- deferred development tasks for the previous mainline (or mainlines). This practice is documented and named as the DROP PASS pattern in [Dikel, Kane & Wilson 2001, p. 91], it allows the organization to *maintain a beat by moving less critical features to later release cycles.* Requirement changes have a similar effect, these are integrated into the delivery system by changing the specification and pushing the work to the next release. As noted in [Lyneis, Cooper and Elsa 2001], mid-project design changes obsolete work already done, whether it had been performed correctly or not. In this context, [Dikel, Kane & Wilson 2001, p. 83] noted that *If it becomes difficult to maintain the rhythm while implementing the key feature, it is most likely a warning sign that the risk and complexity of the feature is greater than anticipated, and that replanning is necessary.*
- deferred defect fixes from previous mainlines (that is, defects that are known on previous releases but that the organization elected not to fix there),
- undiscovered rework from previous development work (note that when some rework is discovered on one mainline, it is then known on all subsequent mainlines, and possibly on previous ones as well).

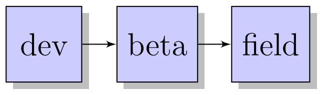
With regards to defect discovery, technical managers have many choices to make: they need to decide what mainlines must be modified to incorporate the fix for a defect, which is usually decided on the basis of severity and customer exposure. For instance, it is no use fixing a bug that customers cannot see on an old release, for instance; conversely if the bug affects many customers, that can warrant an emergency merge of the fix on many parallel branches.

What we want to demonstrate here is that to counter the effects of firefighting, the manager will have to limit the number of currently active branches. The more branches are active, the more opportunities for firefighting to be necessary. Similarly, nominating different people for different branches may have some benefit (although intuitively, having different people doing the same work on different branches is duplication of work, and also nominating some people to do only maintenance work is usually bad for morale).

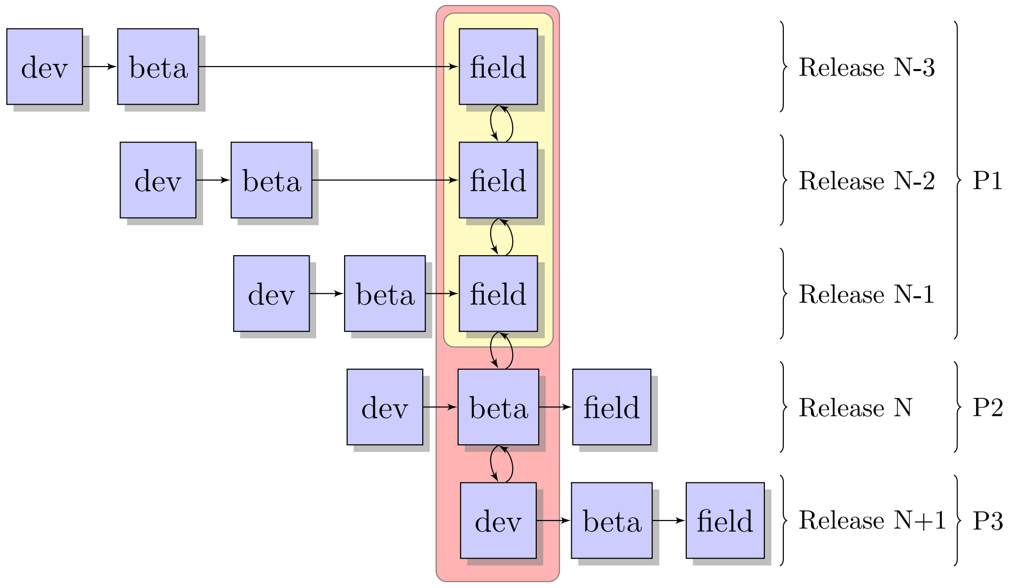
[Abdel-Hamid & Madnick 1989, p. 39] indicated that Nay [Nay 1965] and Kelly [Kelly 1970] have produced system dynamics studies of project management applied to multi-project environments. The present thesis builds on that work in the context of product lines.

5.7 The Rework Cycle in presence of multiple stages on multiple active branches

Releases, like projects, are time-boxed, and each release looks conceptually as in Figure 5-10a. However, as the organization fields a new release of its product family to its customers periodically, multiple releases cycles occur in staggered fashion, and are in fact happening simultaneously (in the time-slice highlighted in red), as shown on Figure 5-10b. This figure depicts four parallel releases, the first three are fielded and used in production by customers(highlighted in yellow in the figure), one is in beta stage, and the last and current one is in full-fledged development mode. Notice how all past releases accumulate in the *fielded* stage. This explains why looking at multiple active branches, each in a given stage of the development lifecycle, is one way to model the development of the product.



(a) Multiple stages on a single branch, as in Section 5.5



(b) Multiple stages on multiple parallel branches, with simultaneous stages highlighted in one time-slice. Highlighted in yellow are all the *fielded* releases, which can be aggregated together.

Figure 5-10: Difference between single-mainline and multiple mainlines.

Also one can readily see depicted in Figure 5-10b the phenomenon of cross-release rework propagation discussed in 5.6 and shown on Figure 5-9. This phenomenon is one particular instance of the *accidental adversaries* archetype (See Section 5.6 and the mapping between the two on Figure 5-11), which accounts for the fact that firefighting on one branch takes a person away from work on another branch, and work on one branch causes additional work to be required on the other. This cross-release phenomenon happens in both directions:

– Work on stage P3 (i.e. the *dev* stage of release N+1, to use the notation from Figure 5-10b, will eventually get transferred to stage P2, when release N+1 becomes the *beta* release on the release date.
– Work on stage P2 causes rework, as any defect fix must be integrated in the future releases, which at that moment in time are represented by the *current* release.
– Work on stage P1 causes rework on P2 and P3 by the same mechanism: bugfixes from incidents reported by the field must be propagated, in other words the organization cannot lose work performed on P1, so it must take some action to propagate that fix (or an equivalent one) to the current stage so that it is included in all future releases.

Node#	Description in Section 5.6	Description in Figure 5-10b
1	A's activity toward B	Forward port of bugfix on the earlier stage to the later stage
2	A's activity toward A	Bugfix to increase quality of one stage
3	B's activity toward B	Bugfix to increase quality of one stage
4	A's success	Quality of one stage
5	B's success	Quality of one stage
6	B's activity toward A	Backport of fix from later stage

Figure 5-11: Mapping cross-release rework onto *accidental adversaries* from Section 5.6

5.8 Multi-Release System Dynamics Model

In this section, we describe a System Dynamics model developed in Vensim® to understand the dynamics of product line development in the presence of multiple released versions of the product family . Large software systems can be developed in two ways:

– either as a collection of subsystems, each built and released independently, then assembled into the larger product system,
– or as one big system where all the components are built and released together.

The first approach, while valid, tends to lead to a combinatorial explosion in the number of combinations of versions of components that need to be tested during system integration testing. That approach could be modeled using System Dynamics as well, but that is not what we want to examine here: we will be looking at a system developed using the second approach described. This approach is used in industry, and lends itself well to the development of large systems, as system integration tested continually.

We start with a description of the model, followed by an explaination of how it describes multiple stages of the software development lifecycle, as well as multiple active mainlines running in parallel. This approach should describe the reality of software development in large software product organizations more closely than the usual static view of a single product development does.

5.8.1 Description of the model

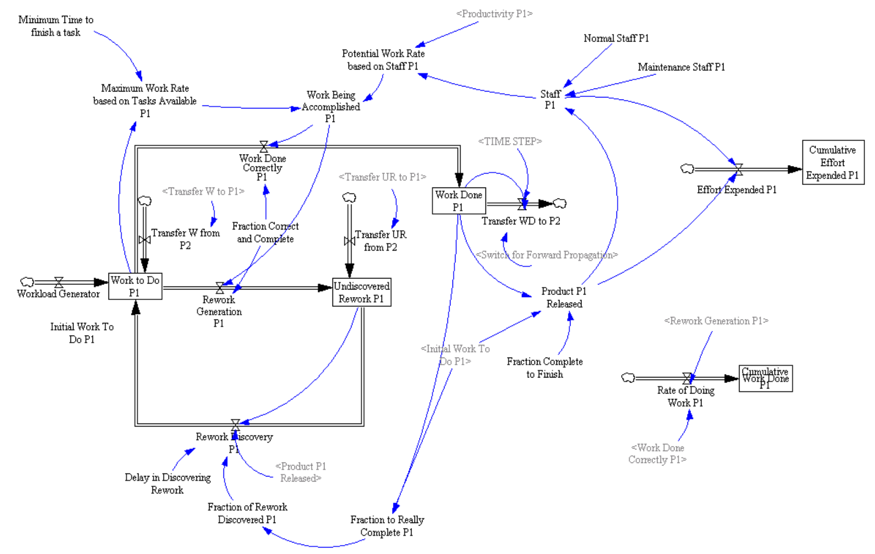
The model presented here is meant to demonstrate the trends that appear in a software product line engineering environment, we know from [Abdel-Hamid & Madnick 1991, p. 177] that portability of results in such models is especially poor, so one would need to re-calibrate model for one's own project. Similarly, [Reichelt & Lyneis 1999] argues that there is no learning across projects: product line engineering presents the best opportunity because there is a guaranteed succession of projects with strong similarity and commonality, where learnings are always applicable to forward releases. In other words, while portability across organizations may lack, portability is present in the case of successive releases within a single enterprise: this where the leverage is present for managers to use. One can collect historical data and then compare with current projects reliably, whereas comparing across companies is an apples-to-orange comparison.

Variable	**Meaning**	**Value**
Release duration	length of one cycle	26 weeks
Simulation duration	10 release periods	260 weeks
nbTasks	number of tasks at release start	1000
FCC	Fraction Correct and Complete	68%
Productivity P1	Productivity of workers in stage P1	0.8
Productivity P2	Productivity of workers in stage P2	0.9
Productivity P3	Productivity of workers in stage P3	1.0
Delay in Discovering Rework	-	15
Minimum Time to finish a task	-	1
Switch for Forward Propagation	boolean to enable propagation of rework across releases	1
Switch for pulse on Release # 5	boolean to simulate additional 50% workload on release #5	0

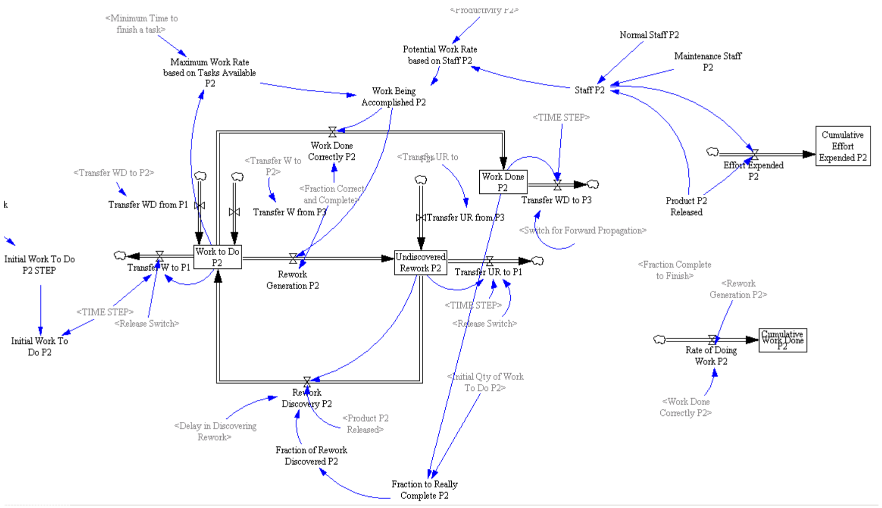
Figure 5-12: Variables and values used in the System Dynamics model from Sections 5.8.1.1 to 5.8.1.7.

We have developed a simplified System Dynamics model of this phenomenon, Figure 5-10 shows the basic states, Section 5.8.1.1 shows the details of the model. The model has three stages, as seen highlighted in Figure 5-10b, named P1, P2 and P3. Those names were chosen for notational convenience in VENSIM, P1 is the earliest stage in the timeline, P2 comes next and P3, as the *current* stage, comes last:

- the *current* stage (also called P3 in Section 5.8.1.3), which models the current model-year in Repenning's terminology. This represents the active under-development mainline. When the release date is reached, all remaining *Work To Do* is transferred to the *beta* stage.
- the *beta* stage (also called P2, shown on Figure 5-13b), which model the mainline currently undergoing Quality Assurance testing and/or preliminary customer testing (hence the name). When the release date is reached, all remaining *Work To Do* is transferred to the *fielded* stage.
- the *fielded* stage (also called P1, shown on Figure 5-13a), which models in aggregate all past-but-still-active mainlines (this represents the stages highlighted in yellow in the Figure 5-10b). These are mainlines that have not been discontinued yet, as customers are still using them in production and require continued maintenance. These see an influx of workload coming from defect reports from the field.



(a) The *fielded* stage (P1, see Section 5.8.1.1)



(b) The *beta* stage (P2, see Section 5.8.1.2)

Figure 5-13: Rework cycles in the first two stages (see complete model in Section 5.8.1.1)

Note that this assumes that all products in the product family are released simultaneously. This is one possible managerial choice, which simplifies system integration testing as it is done continuously up until release time. Another choice is to release each product on its own schedule, and then test its integration with other pieces of the overall system: that choice increases the amount of testing required, we will not be looking at how this can be managed here.

In this model, we have assumed that for each release there is the same number of tasks related to new feature content. The model was calibrated with the sample values shown in Figure 5-12, after our own experience; we expect the reader to adapt the values selected to his own circumstances. In other words, the assumption is that the enterprise knows how to estimate and then limit the amount of tasks to match the capacity of its own software production system, having learned from experience how much can go into any single release. This is an approximation, as it is widely known that project size estimation is very difficult, and sometimes pointless as the real number of tasks to be done changes during the execution of the project. Another simplification is that we model work transfer between stages to happen exactly at the release date: in reality, in the weeks prior to the actual deadline date, the engineering managers distinguish between:

– tasks related to the initial feature set for the current release: these can be pushed to the next release if not done in time,
– tasks related to rework, which ones want to see finished before releasing.

In other words, in reality, managers can and do make adjustments in flight to the stock of *Work To Do P3*.

As depicted on Figure 5-9, there is propagation of work-to-do across mainlines:

- backwards propagation, which occurs when current engineering discovers and fixes defects on the *current* release, and management elects to make the corresponding change on earlier releases.

- forward propagation, which occurs when a defect is found in the field on one of the fielded releases, and the corresponding fix must be propagated to all subsequent releases. It may happen in this case that the fix on a later release is different, because the area of the code has changed or been otherwise rewritten or replaced.

The tasks generated by rework on older releases almost always correspond to new tasks on newer releases, save for the odd case where the component in need of rework has been deprecated, removed, or rewritten (i.e. in this case, there is no point in merging the changes onto newer releases). The tasks generated by rework on newer releases do not always contribute new work-to-do on older releases: that is the case when the component involved did not exist yet, or the defect is not important enough to warrant fixing on older releases.

There is an overall rhythm dictated by the *Release Counter* variable, and when the *current* release reaches its end date, remaining work to do is pushed down to the *beta* stage, *beta* work left is pushed down to the *fielded* stage, and a new set of work-to-do tasks appears for the new release cycle. Each stage also computes *Cumulative Effort Expanded* and *Cumulative Work Done*, which are used for monitoring.

The following subsections show the causal loop diagrams used in our System Dynamics model. The VENSIM model is divided in *views* to afford us modularity which helps keep the model comprehensible. Each view is presented in turn, with annotations to help the reader understand the model.

- Section 5.8.1.1 shows the *fielded* stage (P1), where the *Work To Do P1* stock is connected to *Work To Do P2* on stage P2, and sees a transfer of work at the release anniversary date. Similarly, *Undiscovered Rework P1* is connected to *Undiscovered Rework P2* on stage P2, and sees a transfer of stock at the anniversary date: this models the fact that the *beta* release becomes *fielded* on that date, and therefore all undiscovered issues in that release move with it into the *fielded* status.

- Section 5.8.1.2 shows the *beta* stage (P2), where the *Work To Do P2* stock is connected to *Work To Do P3* on stage P3, and sees a transfer of work at the release anniversary date. Similarly, *Undiscovered Rework P2* is connected to *Undiscovered Rework P3* on stage P3, and sees a transfer of stock at the anniversary date: this models the fact that the *current* release becomes *beta* on that date, and all its issues, known or unknown, are moved into *beta* status.

- Section 5.8.1.3 shows the *current* stage (P3), where the *Work To Do P3* stock sees a flow of tasks at each release anniversary date: the value of this stock is transfered to *Work To Do P2* and simultaneously fed a new set of tasks, marking the start of a new development period. As on previous stages, *Undiscovered Rework P3* is transfered into *Undiscovered Rework P2* at the end of period.

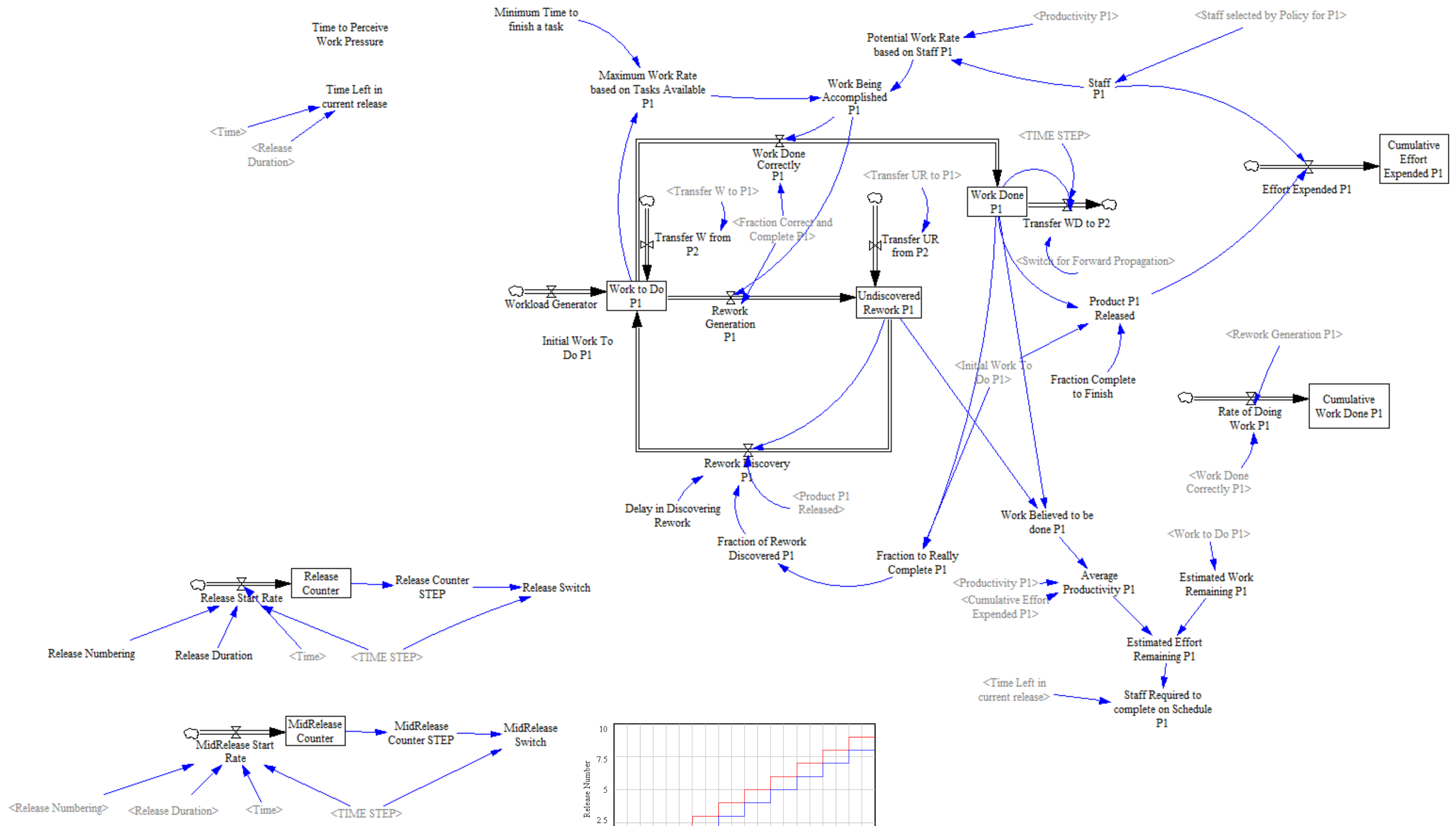
- Section 5.8.1.4 shows the *Productivity*, we set different productivities for the three different stages.

- Section 5.8.1.5 shows the *Staff Allocation Policies*, which we will experiment with in Section 5.8.4. These policies are based on the notion of *staff pressure* (which we also call more simply *pressure*), which is the ratio on any stage of the staff currently allocated to the staff estimated to be required to finish the *Work To Do* within the release cycle. A stage which has a larger backlog of tasks exhibits a greater *pressure* than a stage with a smaller backlog: this is designed to simulate the pressure the manager feels to allocate staff to that one stage. In Section 5.8.4, we experiment with a variety of policies detailed further in that section.

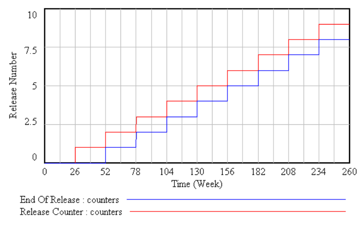
- Section 5.8.1.6 shows the *Fraction Correct and Complete* variables, one for each stage. Each has a nominal value, and then is influenced by the work performed so far in one release cycle: this models the fact that as defects are generated in some portion of the work done, later tasks are more prone to generating new errors on already incorrect work.

- Section 5.8.1.7 shows the inputs and outputs, including the harness used to generate the workload in the various experiments in sections 5.8.4 and later.

5.8.1.1 View 1 : Fielded Releases Stage (P1)

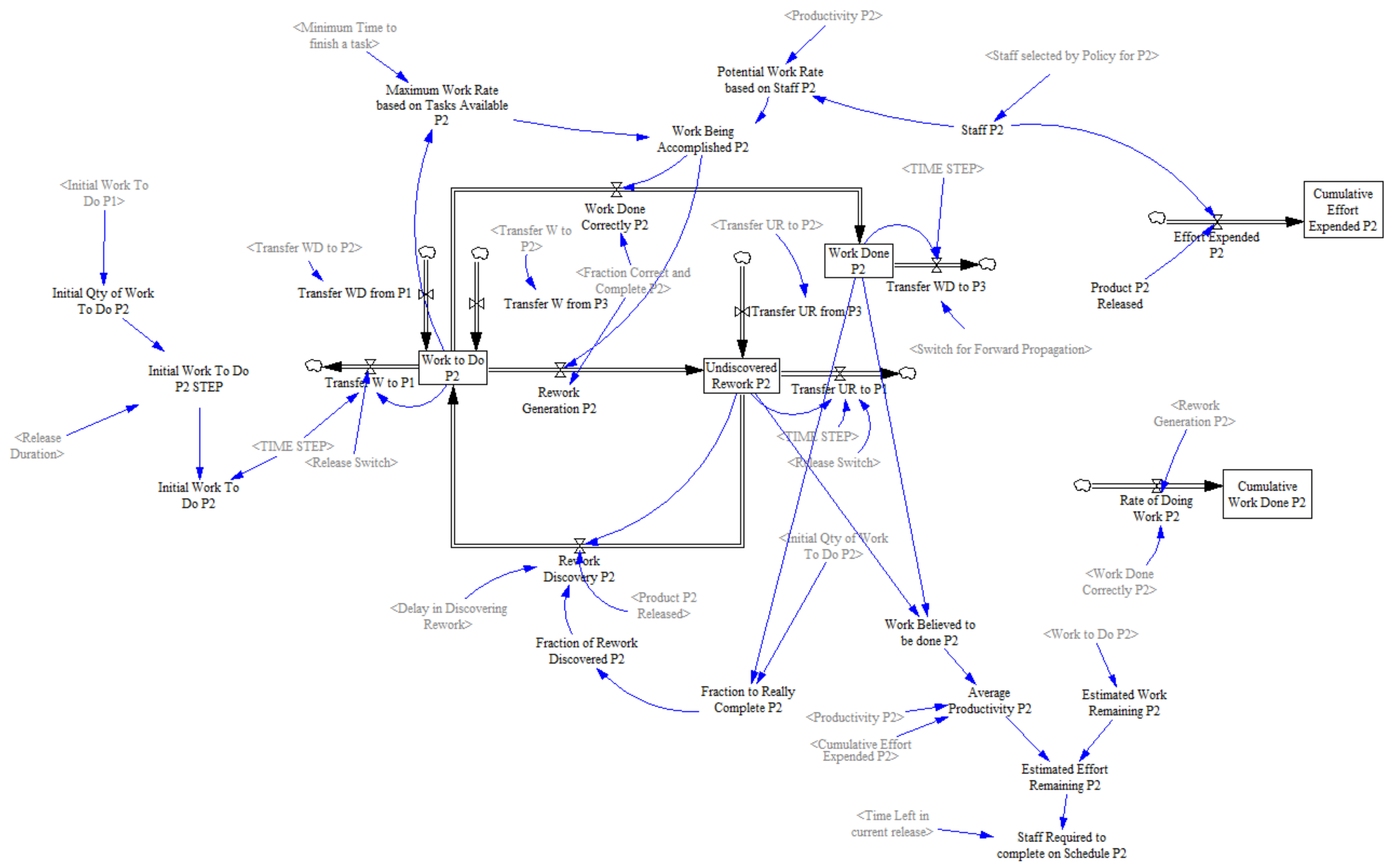


Release Events



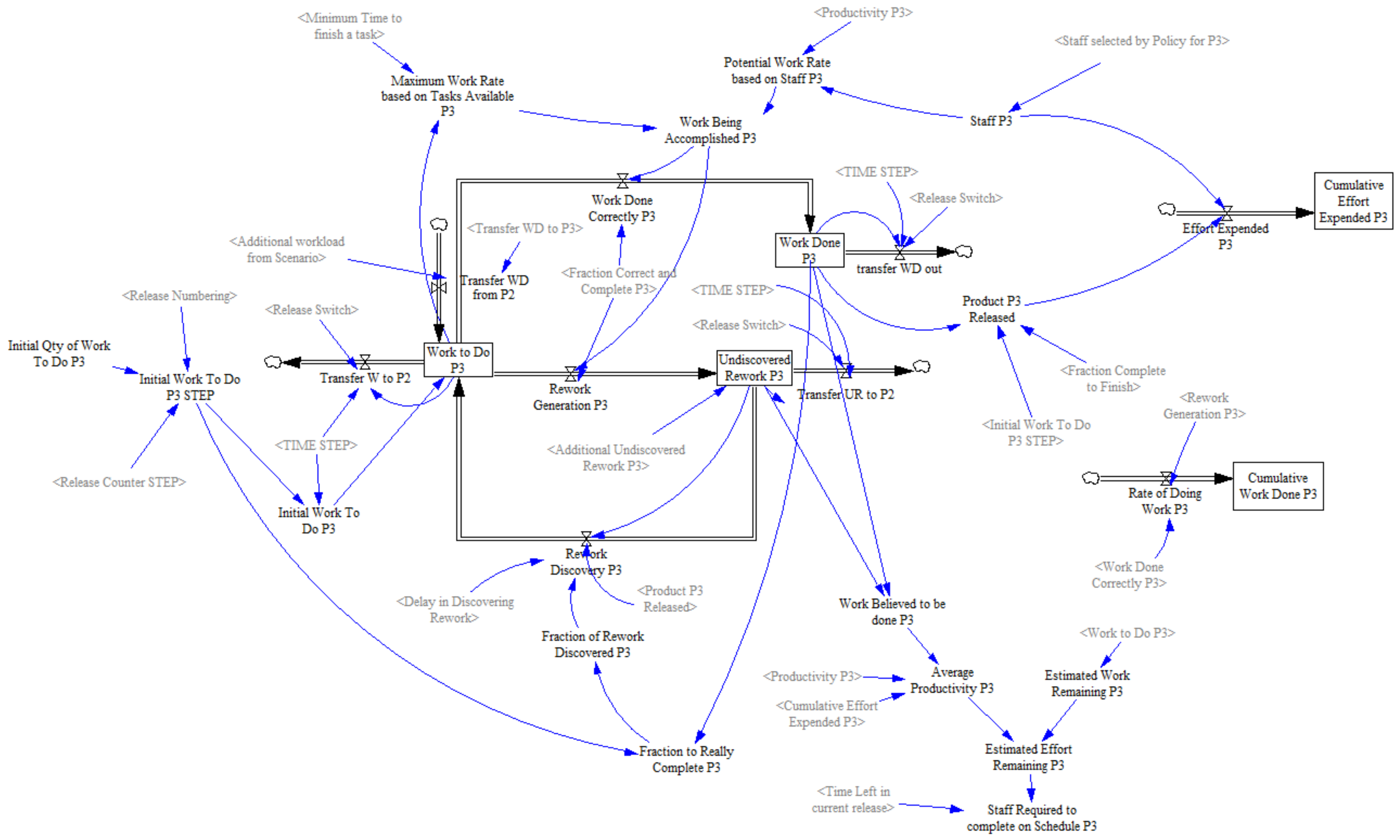
Plot of Release Events

5.8.1.2 View 2 : Beta Release Stage (P2)



<Minimum Time to finish a task>
Maximum Work Rate based on Tasks Available P2
<Productivity P2>
Potential Work Rate based on Staff P2
<Staff selected by Policy for P2>
Staff P2
Work Being Accomplished P2
Work Done Correctly P2
<TIME STEP>
Cumulative Effort Expended P2
Effort Expended P2
Product P2 Released
<Initial Work To Do P1>
Initial Qty of Work To Do P2
<Transfer WD to P2>
Transfer WD from P1
<Transfer W to P2>
Transfer W from P3
<Transfer UR to P2>
Transfer UR from P3
Work Done P2
Transfer WD to P3
<Fraction Correct and Complete P2>
<Switch for Forward Propagation>
Initial Work To Do P2 STEP
Transfer W to P1
Work to Do P2
Rework Generation P2
Undiscovered Rework P2
Transfer UR to P1
<Release Duration>
<TIME STEP>
<Release Switch>
<TIME STEP>
<Release Switch>
Initial Work To Do P2
<Rework Generation P2>
Rate of Doing Work P2
Cumulative Work Done P2
<Initial Qty of Work To Do P2>
<Work Done Correctly P2>
Rework Discovery P2
<Product P2 Released>
<Delay in Discovering Rework>
Fraction of Rework Discovered P2
Work Believed to be done P2
<Work to Do P2>
Fraction to Really Complete P2
Average Productivity P2
Estimated Work Remaining P2
<Productivity P2>
<Cumulative Effort Expended P2>
Estimated Effort Remaining P2
<Time Left in current release>
Staff Required to complete on Schedule P2

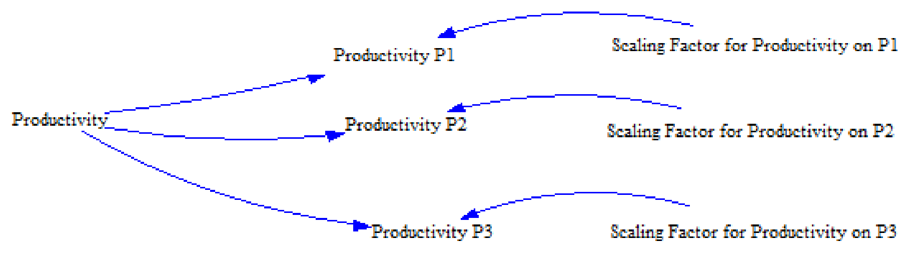
5.8.1.3 View 3 : Current Release Stage (P3)



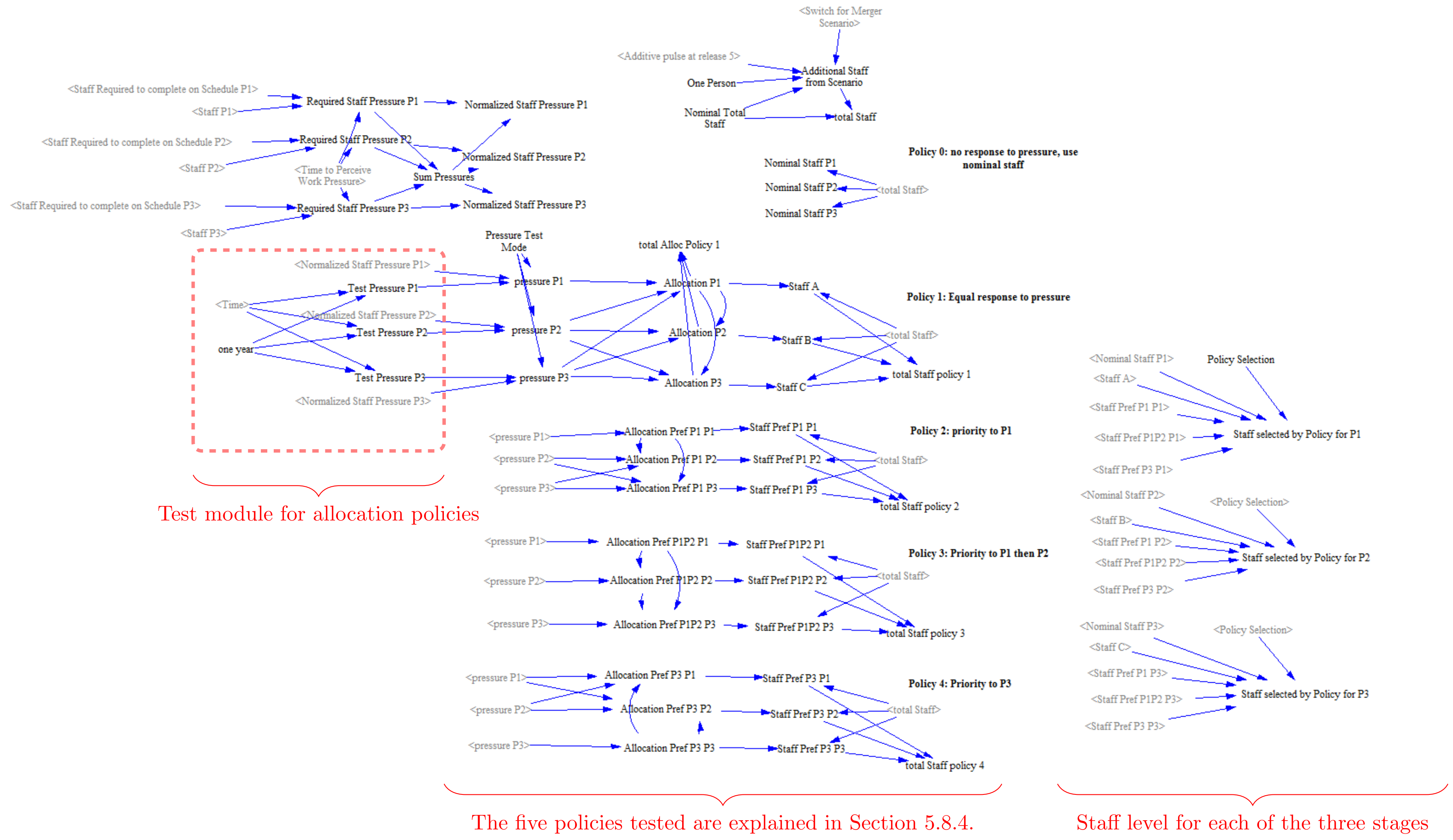
<Minimum Time to finish a task>
Maximum Work Rate based on Tasks Available P3
<Productivity P3>
Potential Work Rate based on Staff P3
<Staff selected by Policy for P3>
Staff P3
Work Being Accomplished P3
Work Done Correctly P3
<TIME STEP>
<Release Switch>
Work Done P3
transfer WD out
Effort Expended P3
Cumulative Effort Expended P3
<Transfer WD to P3>
Transfer WD from P2
<Additional workload from Scenario>
<Fraction Correct and Complete P3>
<TIME STEP>
<Release Switch>
Product P3 Released
<Release Numbering>
<Release Switch>
Initial Qty of Work To Do P3
Initial Work To Do P3 STEP
Transfer W to P2
Work to Do P3
Rework Generation P3
Undiscovered Rework P3
Transfer UR to P2
<Fraction Complete to Finish>
<Rework Generation P3>
<Initial Work To Do P3 STEP>
<TIME STEP>
<Additional Undiscovered Rework P3>
<Release Counter STEP>
Initial Work To Do P3
Rate of Doing Work P3
Cumulative Work Done P3
Rework Discovery P3
<Product P3 Released>
<Delay in Discovering Rework>
<Work Done Correctly P3>
Work Believed to be done P3
Fraction of Rework Discovered P3
<Work to Do P3>
<Productivity P3>
Average Productivity P3
Estimated Work Remaining P3
<Cumulative Effort Expended P3>
Estimated Effort Remaining P3
Fraction to Really Complete P3
<Time Left in current release>
Staff Required to complete on Schedule P3

5.8.1.4 View 4 : Productivity

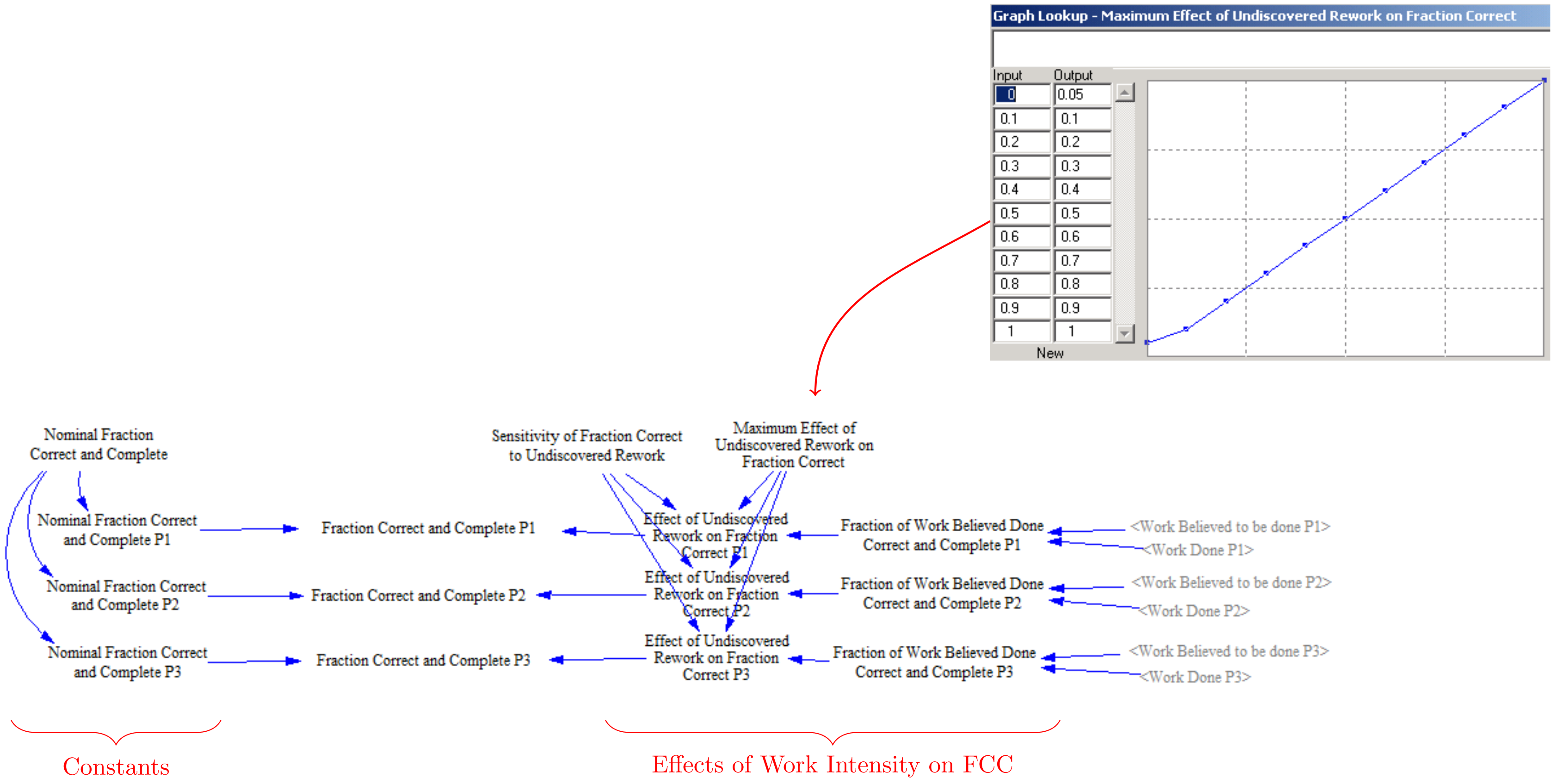
This view shows how Productivity is computed for each of the stages, we have one master constant that feeds into one for each stage, with a multiplicative scaling factor to account for the differences in the nature of the work on each stage.


Productivity P1
Scaling Factor for Productivity on P1
Productivity
Productivity P2
Scaling Factor for Productivity on P2
Productivity P3
Scaling Factor for Productivity on P3

5.8.1.5 View 5 : Staff Allocation Policies

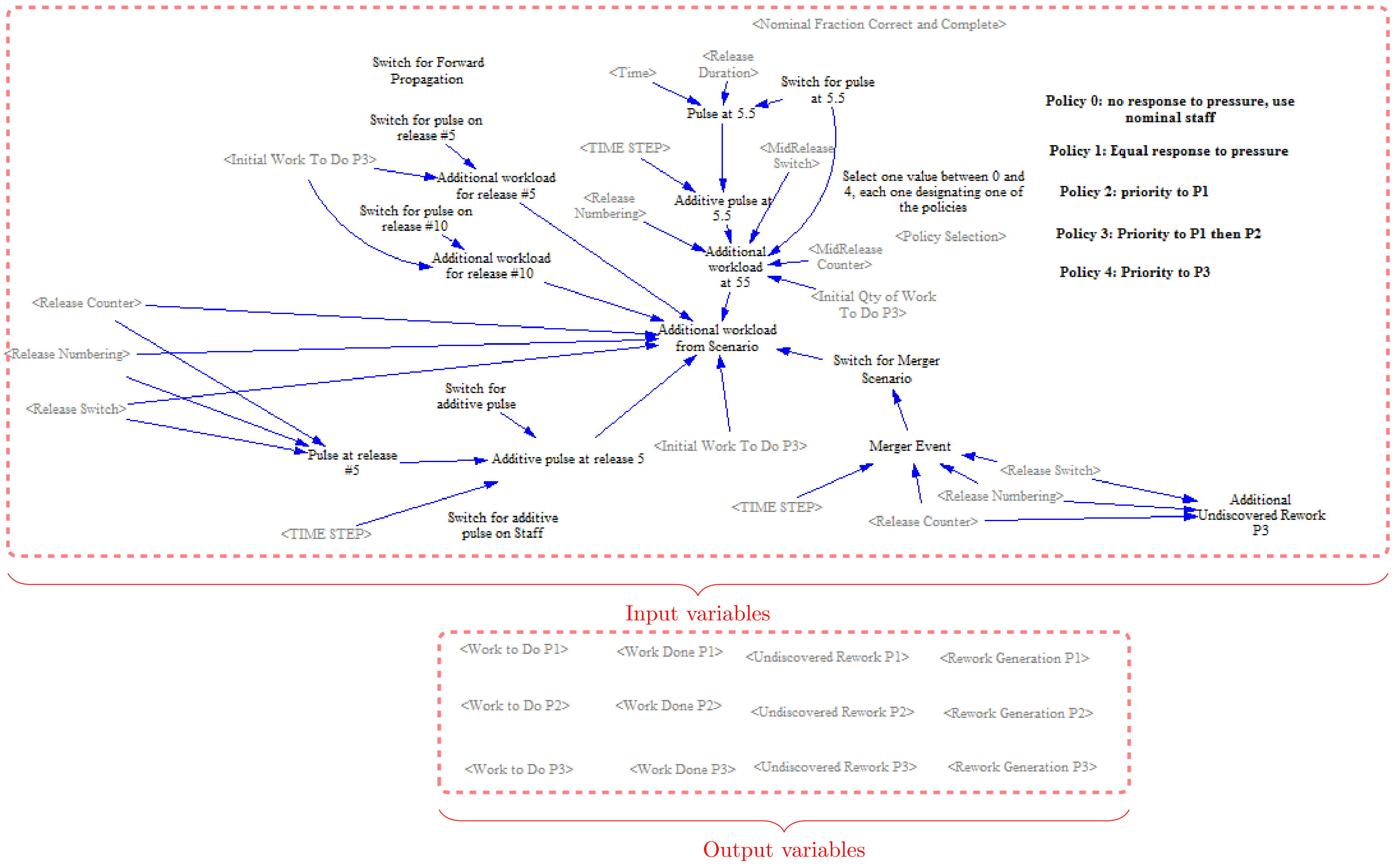


5.8.1.6 View 6 : FCC


Graph Lookup - Maximum Effect of Undiscovered Rework on Fraction Correct
New
Nominal Fraction Correct and Complete
Nominal Fraction Correct and Complete P1
Nominal Fraction Correct and Complete P2
Nominal Fraction Correct and Complete P3
Fraction Correct and Complete P1
Fraction Correct and Complete P2
Fraction Correct and Complete P3
Sensitivity of Fraction Correct to Undiscovered Rework
Maximum Effect of Undiscovered Rework on Fraction Correct
Effect of Undiscovered Rework on Fraction Correct P1
Effect of Undiscovered Rework on Fraction Correct P2
Effect of Undiscovered Rework on Fraction Correct P3
Fraction of Work Believed Done Correct and Complete P1
Fraction of Work Believed Done Correct and Complete P2
Fraction of Work Believed Done Correct and Complete P3
<Work Believed to be done P1>
<Work Done P1>
<Work Believed to be done P2>
<Work Done P2>
<Work Believed to be done P3>
<Work Done P3>
Constants
Effects of Work Intensity on FCC

Input	Output
0	0.05
0.1	0.1
0.2	0.2
0.3	0.3
0.4	0.4
0.5	0.5
0.6	0.6
0.7	0.7
0.8	0.8
0.9	0.9
1	1

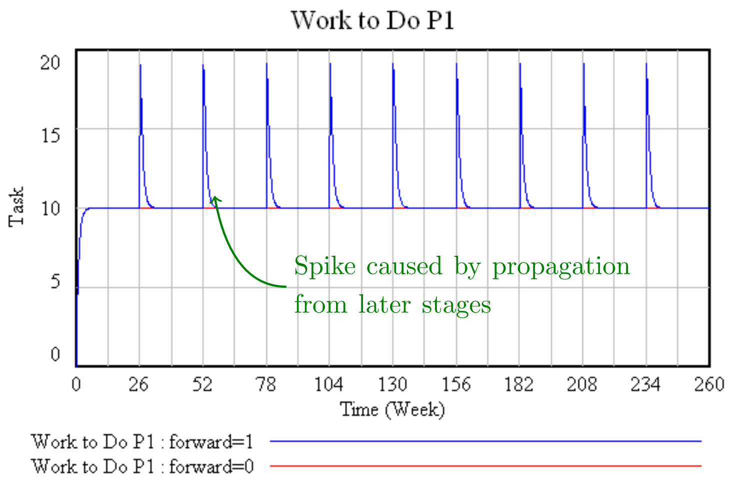
5.8.1.7 View 7 : Summary



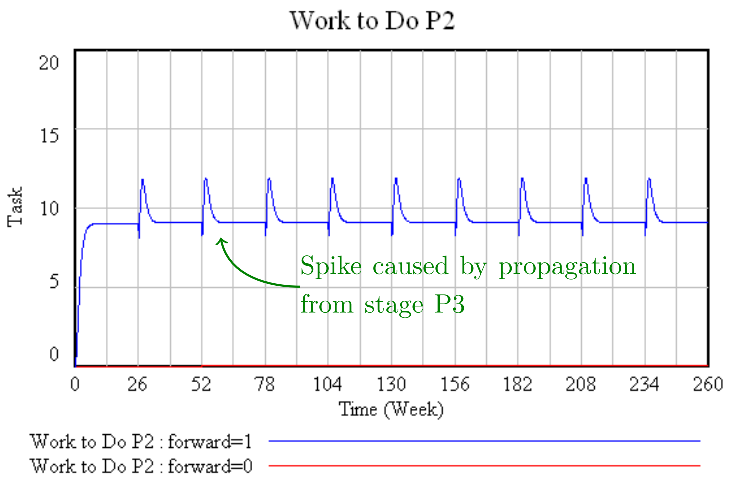
5.8.2 Running the model base case

In this section, we run the model with initial values for all parameters, the goal being to demonstrate its features and how it relates to the three-stage description given of multi-release software engineering. The first experiment is to run the three stages decoupled (depicted in red), and then coupled via the cross-stage rework propagation described in Section 5.7 (in blue).

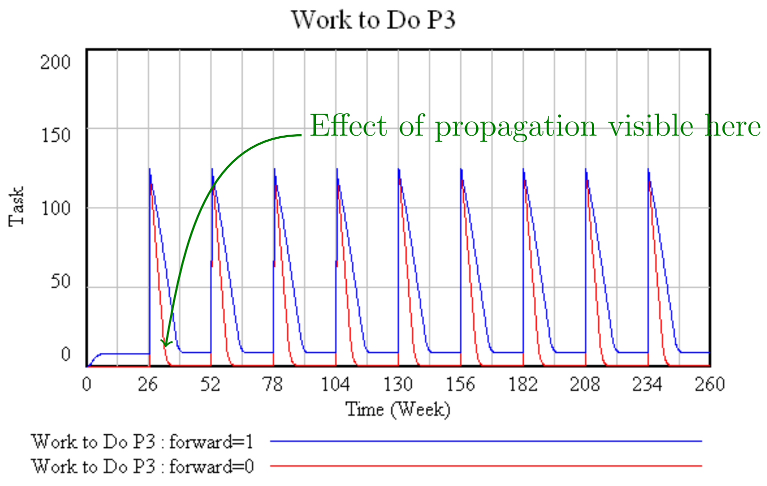
In Figure 5-14b, when propagation is not active, there is no work to do, as expected. Work on this stage only comes from rework propagated from P1 and from work transferred at the release date from P3.



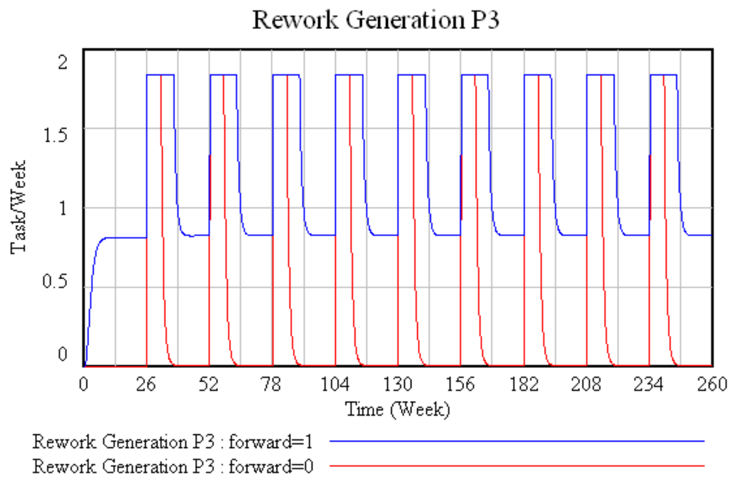
(a) *Work To Do* on stage 1 (*fielded* releases)



(b) *Work To Do* on stage 2 (*beta* release)



(c) *Work To Do* on stage 3 (*current* release)



(d) Rework Generation on stage 3

Figure 5-14: Activating forward propagation shows how rework coming from *fielded* releases increases workload on later releases.

Examining Figure 5-14c further, we can see that steady state is attained after one full release cycle of 26 weeks has passed. We use the term *steady state* as it is used in *thermodynamics* or *engineering* to mean that the system has achieved a predictable state. After a start-up transition on our plots, we see that each release cycle repeats the same pattern; also it is the state we expect the system to return to after being subjected to a change in inputs.

All experiments in this work will be run with a release cycle duration of exactly 26 weeks (except where mentioned otherwise), we assume that management has mandated the organization to produce one new release every 6 months.

Considering the effects of propagating rework between releases causes:

- rework to happen in spikes at the start of the release schedule on *fielded* releases, on Figure 5-14a. The rework spikes in stage P1 comes from later stages, it adds to the constant inflow of rework coming from issues reported in the field,

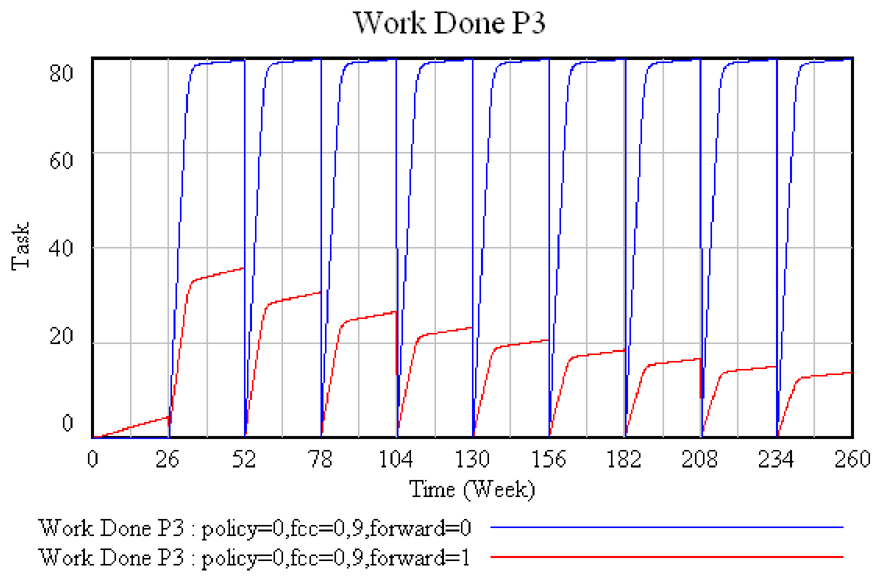
- rework on stage P2, seen on Figure 5-14b, comes mostly from rework propagated from stage P1, with periodic spikes coming from work transfer from P3 at the start of the release schedule (since a *current* release becomes a *beta* one at that specific moment).

- rework on stage P3, seen on Figure 5-14c, increases significantly because of tasks propagated from stages P1 and P2.

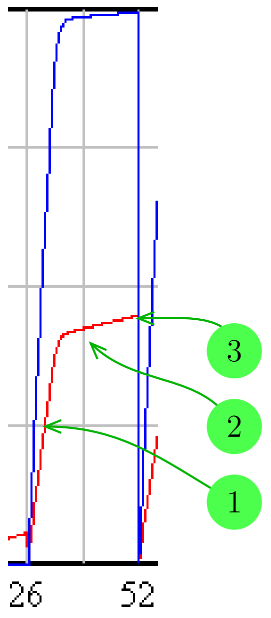
We can see that a model which does not consider propagation of rework from other releases clearly underestimates the amount of work to do at any point in time: Figure 5-14d shows that there is consistenly more rework generated on stage P3 when changes are propagated from earlier stages.

In Figure 5-14 we show the *Work To Do* in each of the three stages for two runs of the model, one ($forward = 0$) does not consider forward propagation of bugs from maintenance of fielded releases, while the other ($forward = 1$) does. The sawtooth profile seen on Figure 5-14c is explained by the fact that the tasks are assigned to the software production system at the exact anniversary date of the release cycle: the first release cycle sees work in the first stages, but not in the *current* stage (also called P3), then cyclically, every 26 weeks, the product management function makes *current* become the *beta* release and creates a queue of work to do for the new *current* stage. Work is then consumed over the duration of one cycle, only to see new work come in from product management at the beginning of the next cycle. We can note that on Figure 5-14c, *Work To Do P3* drops to zero about half-way through the release cycle. This indicates that the system operates well below its capacity, and corresponds to the reality that management allocates some slack directly into the scheduling (i.e. it artificially inflates the stock of *Work To Do*) to account for the inevitable extra work that the organization has experienced.

Work To Do P1 in Figure 5-14a models that the organization is treating the incoming tasks coming from defects reported by the field deployments of the product. After a few release cycles, the system has reached a steady state, with work coming constantly from the field on P1, and work coming at the beginning of each cycle. In the four figures, we depict in red the case when the three stages are independent, save for work transfer at the end of the release cycle, and in blue the case when rework from P1 propagates to P2 and then P3. Figure 5-14c clearly shows that more *Work To Do* remains, thus less work is accomplished on P3 when propagation is active in the model: this represents the fact that staff who would be processing P3 tasks is in fact busy working on P1 and P2, for some portion of the time. This matches our industry experience, where less-than-ideal upfront work always ends in maintenance problems. In addition, when such problems are severe enough, they prevent production use by customers, and therefore warrant immediate attention, another example of *firefighting*.



(a) *Work Done* on stage P3



(b) Detailed view for one cycle

Figure 5-15: *Work Done P3.*

Figure 5-15a displays the behavior of actual *Work Done P3* over time. Figure 5-15b shows the detail of one release cycle, allowing us to look at it more closely:

– 1 is the first phase in the release cycle: work is accomplished at the work rate the organization can sustain, it then reaches a limit because there is no more new *Work To Do*.

- 2 shows the start of the second phase: the organization is performing the rework to fix defects in the work product of the first phase.

- 3 is the end-of-release time, where work for this cycle stops and is transferred to stage P2 (i.e. the *beta* stage). At this time, a new cycle starts anew.

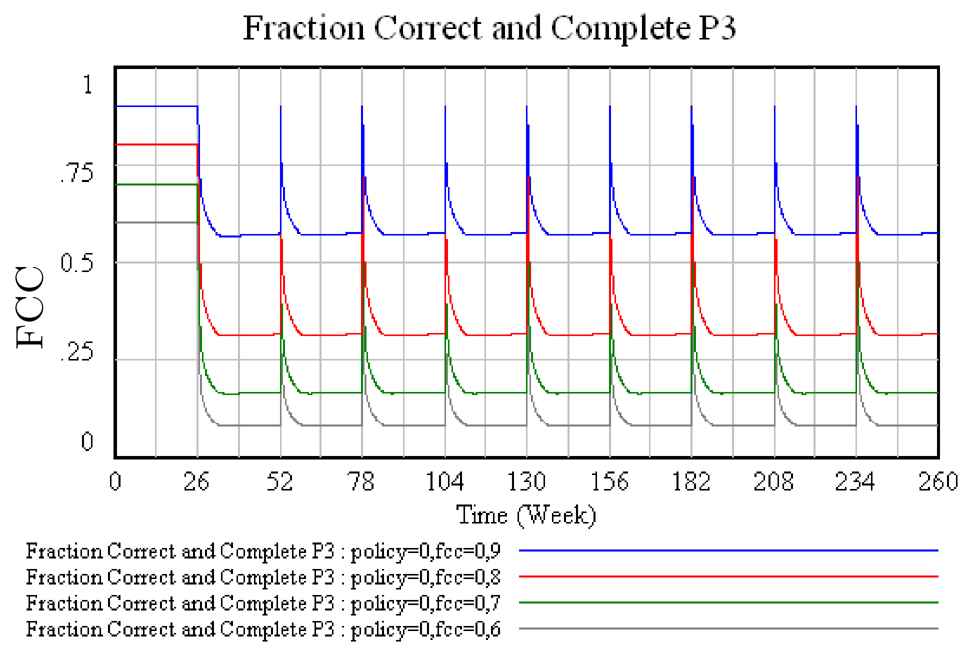
In this model, we ignore the effects of hiring (see Section 3.1.2), and consider the enterprise keeps its staff of engineers on payroll no matter what, so payroll is a sunk cost: we only care about maximizing the return on that investment such that engineer be paid to perform actual work and if possible correct work from the first time, rather than rework caused by upstream errors. We say nothing of cost estimation, we only want to model completion of the project, without regards to actual deadlines: we just want to be able to demonstrate managerial levers that make the project feasible at all and detect conditions that cause unfeasibility, all other things being equal.

This concludes our description of our multi-release System Dynamics of software product line engineering. In the next sections, we will use this model to run various simulations and try to understand the dynamics at play in this kind of software product engineering.

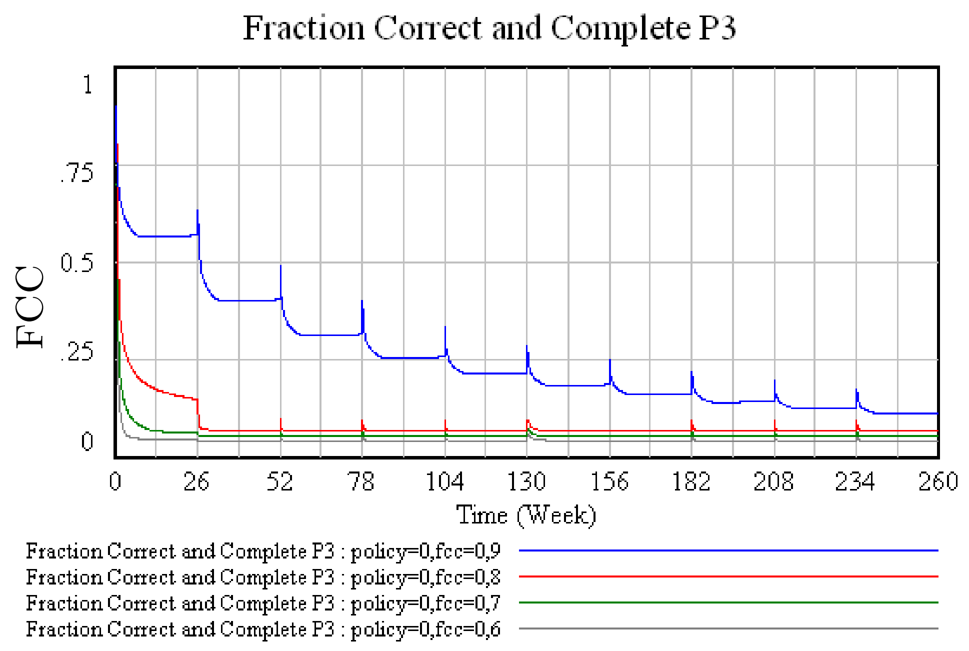
5.8.3 Varying Fraction Correct and Complete

In this section, we want to show how desirable it is to have the highest possible *Fraction Correct and Complete*. [Cooper 1993] found that number to range from 34% for DOD projects to 68% for US commercial software projects: to relate more specifically to commercial software development, we shall use a range of values around 68%. This *FCC* is an abstract construct of the model and cannot be set or known directly. However, intuitively one understands that putting more effort into earlier stages of the Software Development Life Cycle yields better work products and thus less opportunity for errors and errors-on-errors to occur. If a company were to do a post-mortem to estimate FCC, it could possibly apply that value to other projects (assuming all else to be equal: same staff, same kind of project, etc...), or later iterations of the same release cycle. In fact, software product line engineering presents a unique opportunity for this, as successive release cycles have much more in common than development projects might.

Note in Figure 5-16 how the steady state is achieved much earlier and much more favourably when there is no propagation of rework across releases. The *Fraction Correct and Complete* drops at the beginning of each release cycle; in Figure 5-16b, the effect is even more dramatic due to cross-release rework. Note that in this experiment, *FCC* drops to values between 4% and 11%, which is probably not realistic. We do not take those numbers at face value, and are more interesting in the decrease in *FCC* than in actual numbers. This effect can be explained by the amount of work transferred between stages at the release date: as FCC gets lower, the amount of undiscovered rework transferred gets bigger, and that increases the amount of work that will be transferred back to P3. The cross-release propagation indeed increases the amount of work to do: even though large amounts of work get purged at the end of every cycle, much work comes back to the *current* release in the form of propagated rework.

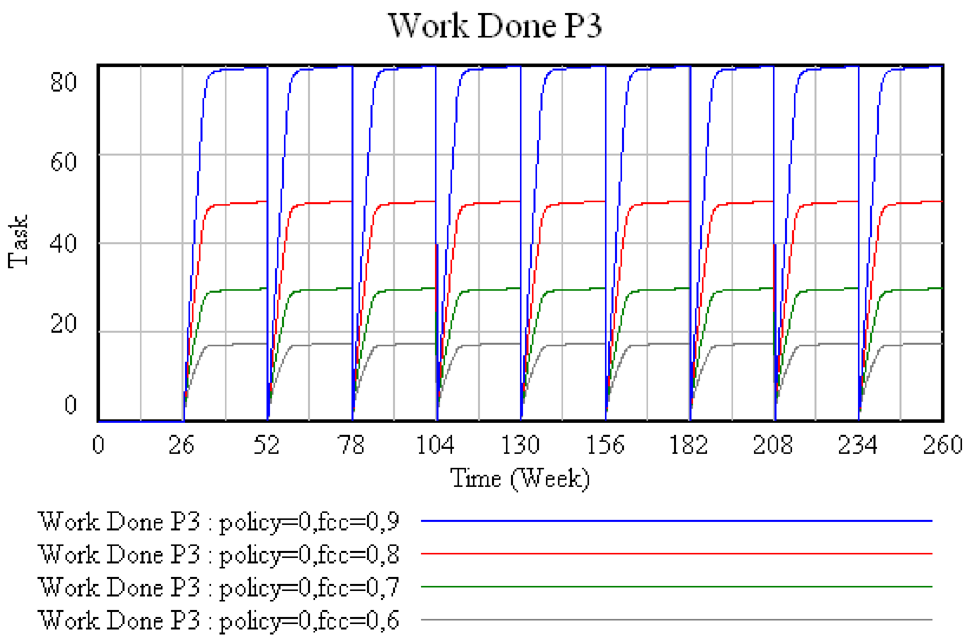


(a) *FCC* on stage 3 without propagation

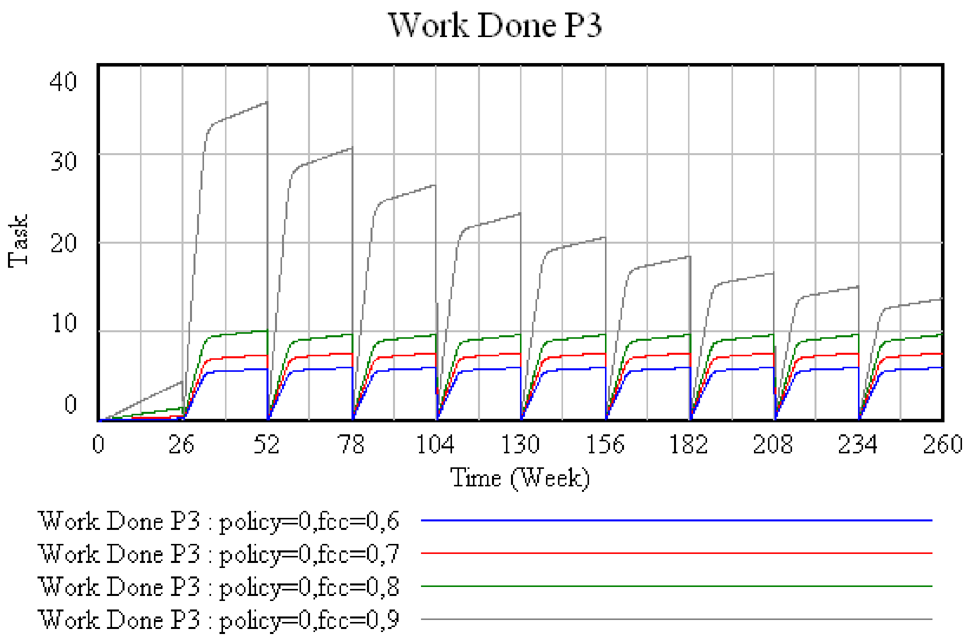


(b) *FCC* on stage 3 with propagation

Figure 5-16: Activating forward propagation shows how *FCC* behaves.



(a) *Work Done* on stage 3 without propagation



(b) *Work Done* on stage 3 with propagation

Figure 5-17: Activating forward propagation shows how *Work Done P3* behaves

In Figure 5-17, the propagation of rework from earlier stages has an adverse effect on the *Work Done* in the *current* stage, even when using separate staff to perform the work on the various stages. Industry experiences have shown us that when maintenance staff is separate from development staff, managers consider that maintenance work has little impact on *current* development work, this experiment would tend to show the opposite is true: maintenance does have a non-negligible impact! This surprising finding tells us that the manager should want to limit the amount of such rework, for instance by limiting the number of active past mainlines.

The lesson here for the manager is that working on multiple stages in parallel causes non-linear effects: even if the work product has good quality, the quality issues are transferred between stages but come back and influence the ability of the organization to produce quality work products. The moral is to *do it right the first time*, otherwise you will have to fix it, and possibly fix it multiple times in all currently active mainlines.

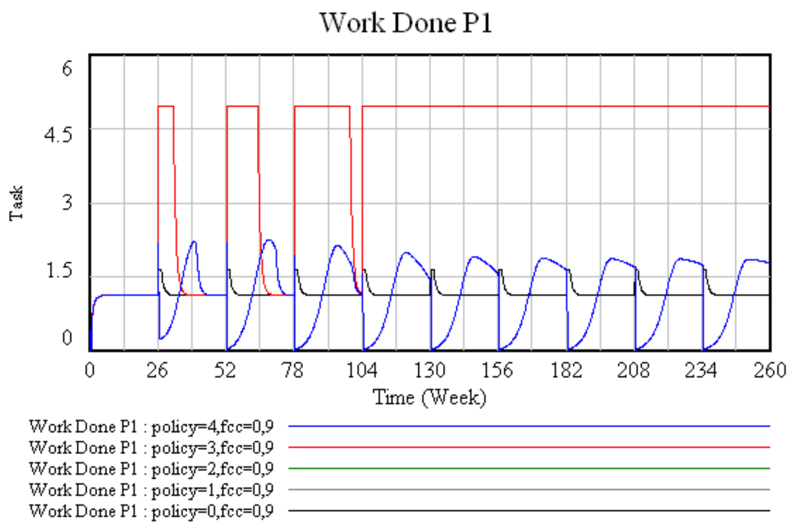
5.8.4 Staff Allocation Policy

In this section, we would like to test the effects of using various policies to allocate staff between the three stages.

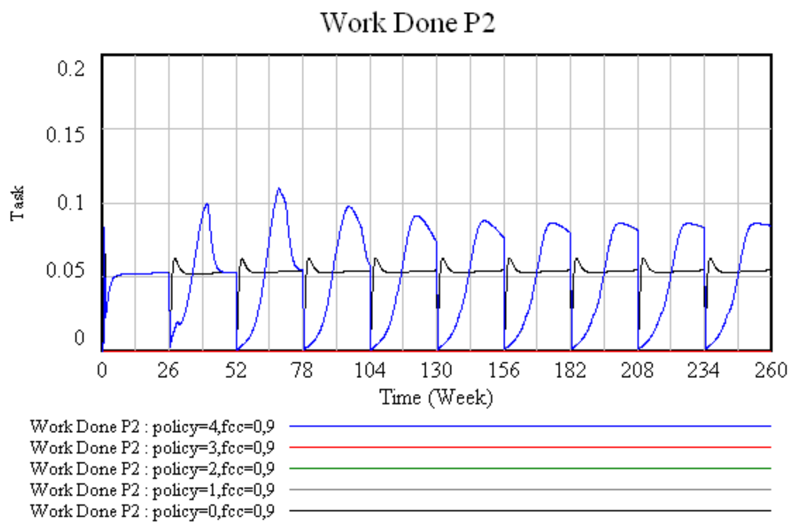
The policies are the following:

- Policy 0 allocates a fixed staff to each stage. This ignores stage pressure, and each stage operates independently: they do not share staff.
- Policy 1 is designed to respond to pressure in the same amount on every stage, so a stage with higher pressure will get priority. This is designed to simulate an immediate firefighting policy that a manager could have: as emergencies come up, staff is immediately allocated to the corresponding stage.
- Policy 2 gives priority to stage P1, then gives equal priority to stages P2 and P3. This simulates the policy of always fighting emergencies coming from the field, but provide equally for other stages.
- Policy 3 gives priority to stage P1 then stage P2, and stage P3 is treated last, with any left over staff that previous stages have not used up.
- Policy 4 gives priority to stage P3, then gives equal priority to stages P1 and P2.

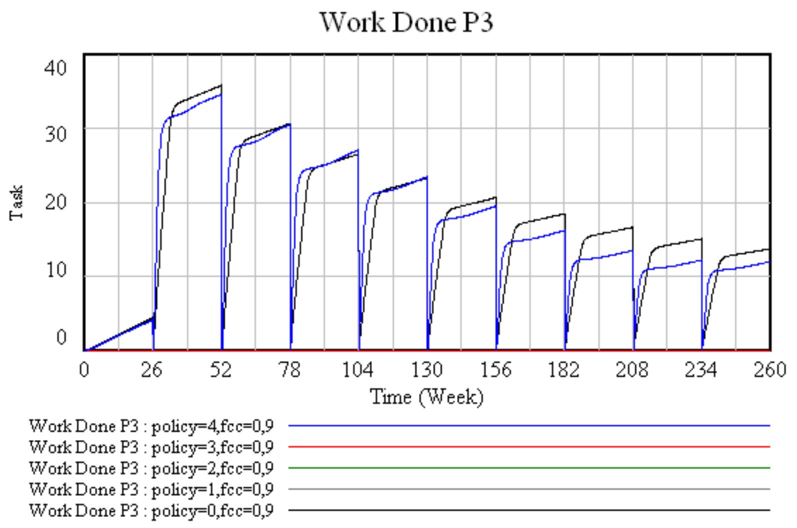
Policy 0 simulates a single stock of staff, while the other policies use three separate stocks, one for each stage. we ran the simulation for each of the allocation policies, and it yielded the following plots for *Work Done*:



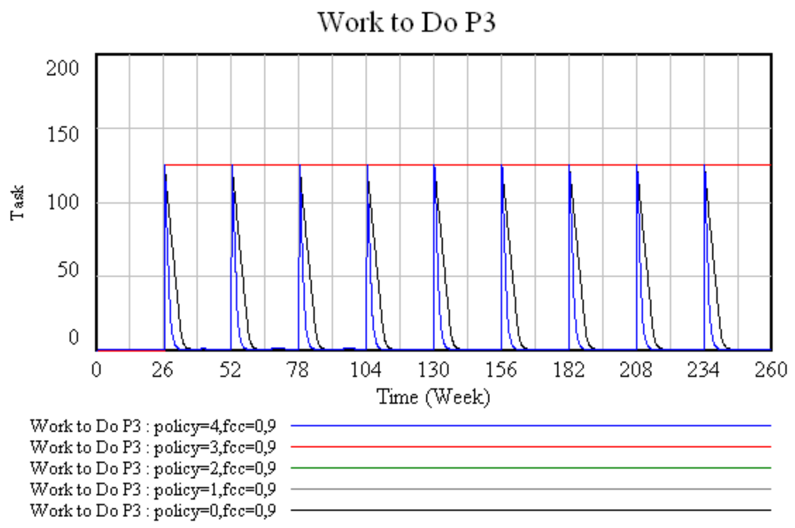
(a) Work Done on stage P1 (the green line is the same as the red one, thus is invisible)



(b) Work Done on stage P2



(c) Work Done on stage P3



(d) Work To Do on stage P3. Notice that policy 3 prevents work from advancing on stage P3.

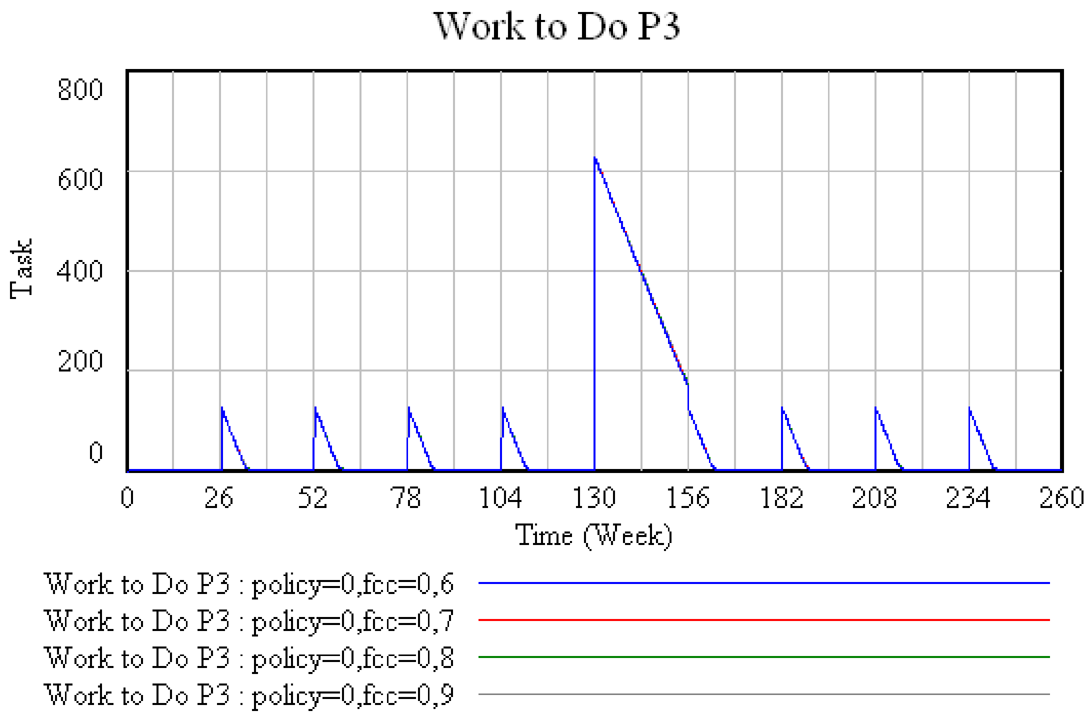
Figure 5-18: Behavior of *Work Done* under various staff allocation policies

One thing of notice here is that for policies 2 and 3, which give priority to P1, if there is enough work to do on stage P1 the pressure on P1 will increase, and if it becomes greater than the pressure on other stages, it will consume all available staff and leave nothing for the other ongoing release trains. For instance, note how *Work To Do P3* remains constant on Figure 5-18d: this means that no work gets done on stage P3, it has been starved by the earlier stages. In a more realistic setting, management might decide to suspend work on the other release cycles, stop all development work until the amount of work on *fielded* releases comes back to acceptable levels: this is the essence of *firefighting.* The lesson here for engineering managers is that one needs to provide some level of isolation between parallel releases, otherwise emergencies on one of them can consume all the resources of the organization, which will block *current* development work.

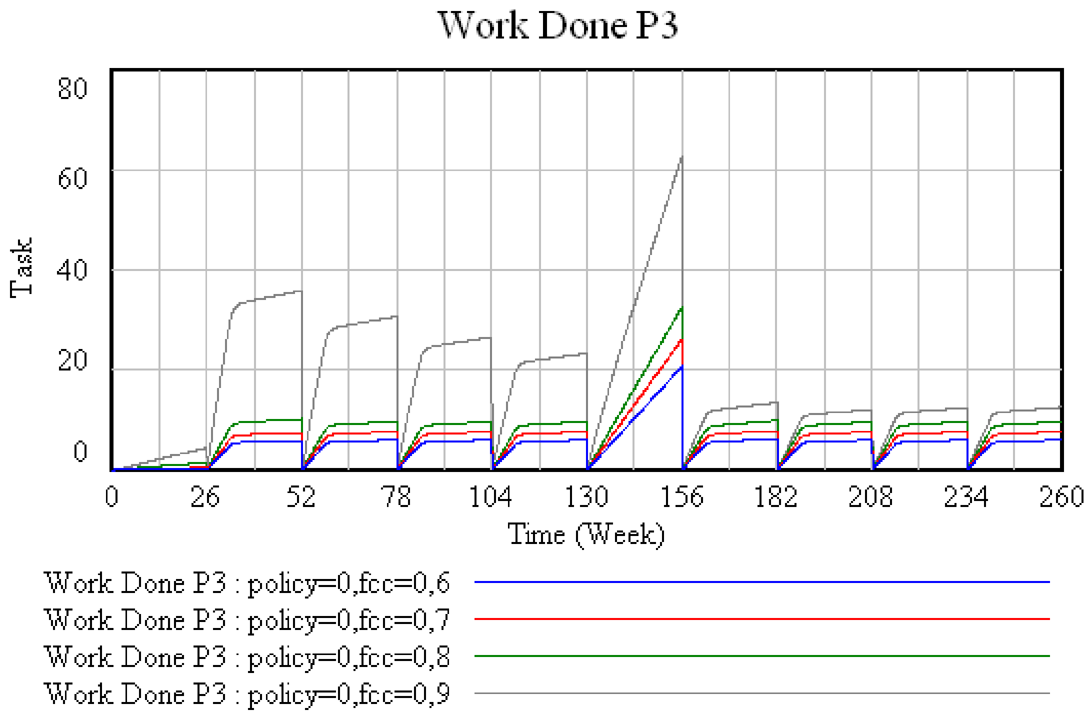
On Figure 5-18d and Figure 5-18c, note how policies 0 and 4 are marginally different, with the effects of sharing staff (policy 4) being visible: even though the organization has selected to give P3 (the *current* release cycle) priority, it accomplishes less work overall when the resources are shared with other parallel release cycles. This has management implications, for there is a tension between maintaining separate staff for maintenance and development, and sharing it throughout. On the one hand, staffing a separate team for maintenance may be difficult to do, as individuals might not enjoy maintenance work, or may not have acquired the knowledge necessary to accomplish their work effectively. On the other hand, sharing staff runs the risk of seeing the development capability erode as maintenance workload fluctuates.

5.8.5 System Behavior under one-time workload increase

In this subsection, we want to examine what happens to the software production system if at some point in time, the engineering managers decide to increase the workload on a given release cycle. To perform this experiment, we added a pulse during release #5. Note that the fifth release cycle was selected for experimentation such that the system has reached a steady state before changing inputs. The now familiar sawtooth shape indicates burst of incoming work at release start; the increase of work at week 130 is clearly visible:



(a) *Work To Do* on stage 3



(b) *Work Done* on stage 3

Figure 5-19: Shows (for various values of *FCC*) how *Work Done P3* behaves when propagation is active and the workload on release #5 is increased

Figure 5-19a presents the change in input for this experiment, while Figure 5-19b shows what changed in the output as a result of the change in input, plotted for several values of *Fraction Correct and Complete*.

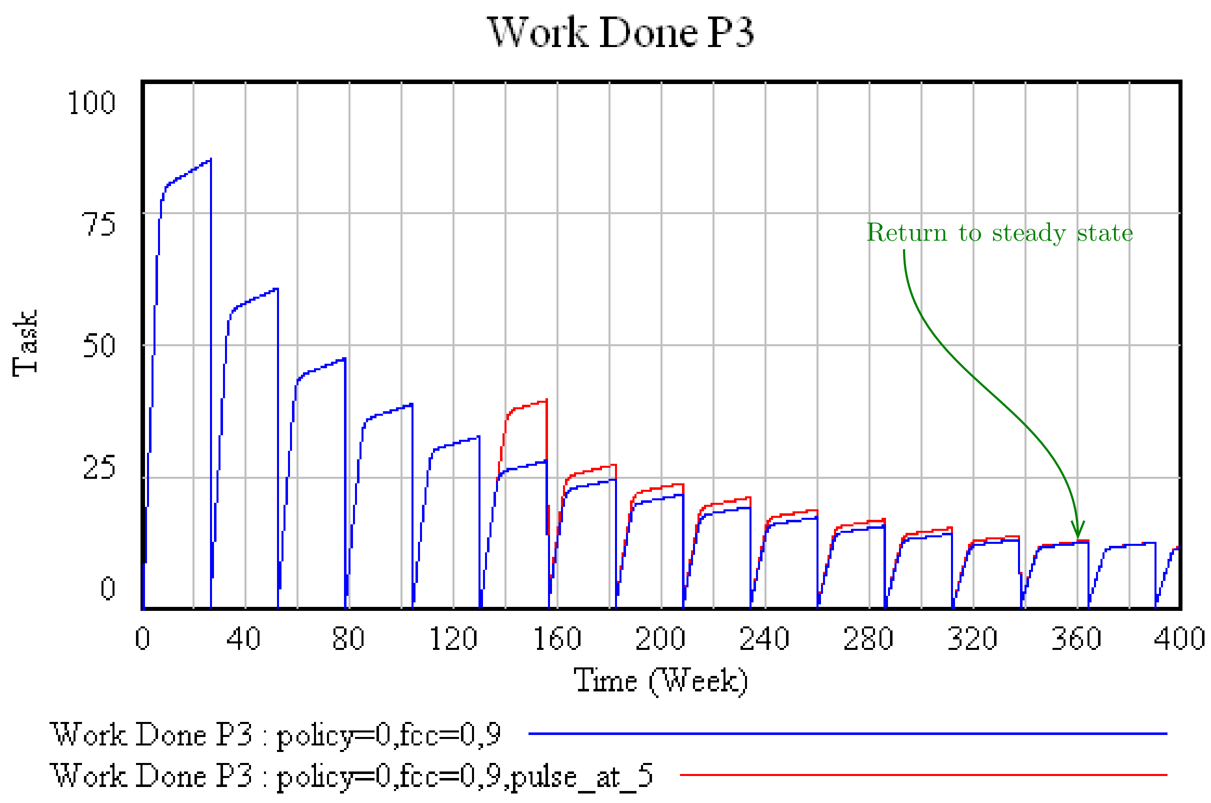


Figure 5-20: *Work To Do* on stage 3

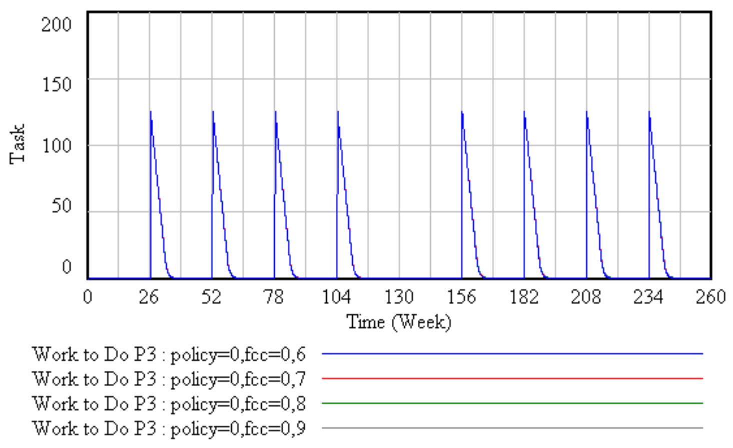
If we compare Figure 5-19b and Figure 5-17b, also shown on the same plot on Figure 5-20 (for FCC=0.9 and 15 cycles), we can see that even such a short-term change can have long-lasting effects: a large burst in work performed immediately in the release cycle seeing the change in workload, followed by an effect on subsequent release cycles, taking more than thirteen cycles to return to the steady state! The system may be operating under capacity and thus can give a lot of immediate results to try and catch up with the increased load, but the adverse effects on the output capability of the system are visible for a long time. In other words, if the manager tries to squeeze too much work into a time-boxed release, the amount of post-release work this change generates on *beta* and then *fielded* stages , and consequently on the *current* stage, exerts the software production system for a long time.

The lesson here is that management should think careful about attempting to force feed too much work into the system: even a healthy production system may take a long period to recover from the effects of such a decision.

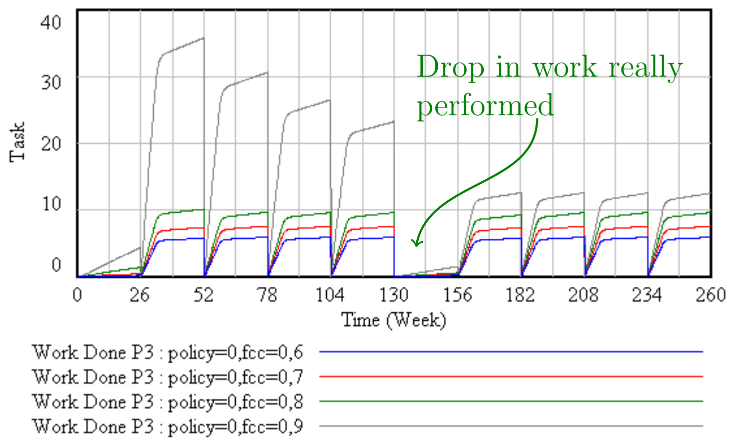
5.8.6 System Behavior under workload decrease

In this subsection, we experiment with a drop in workload during release #5. The decrease of work at week 130 is clearly visible on Figure 5-21a. The experiment was run for values of FCC in $\{0.6, 0.7, 0.8, 0.9\}$, and Figure 5-21c shows the difference for FCC=0.9 between the normal workload and the decreased workload at release #5.

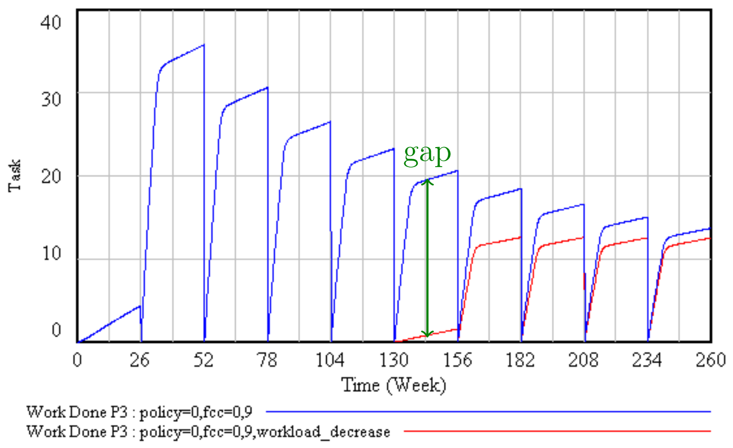
Figure 5-21a presents the change in input for this experiment: the drop in workload is visible for release #5. Figure 5-21b shows the corresponding change in output, with a drop in work performed at the same time. Figure 5-21c plots our baseline case (in blue) together with the case at hand (in red), the gap in work accomplished is visible, but we can also see that after a couple of release cycles, the *Work Done P3* quickly gets back to its usual level, faster than the system recovered from an increase in workload in Section 5.8.5. This prompts us to run the next experiment combining decrease and increase in workload, as the manager might like to know if he can trade a workload decrease at some time for an increase at some other time.



(a) *Work To Do* on stage 3



(b) *Work Done* on stage 3



(c) Comparison of normal workload with temporary decrease at release #5, at FCC=0.9

Figure 5-21: Shows how *Work Done P3* behaves when propagation is active and the workload on release #5 is zero.

5.8.7 System Behavior under workload variation

In this subsection, we try to combine the effects of previous experiments 5.8.6 and 5.8.5 on varying the workload. Here we simulate two changes in workload: one at release 5 and another at release 10, and the length of the simulation is set at 15 years to visualize long-term effects, if any.

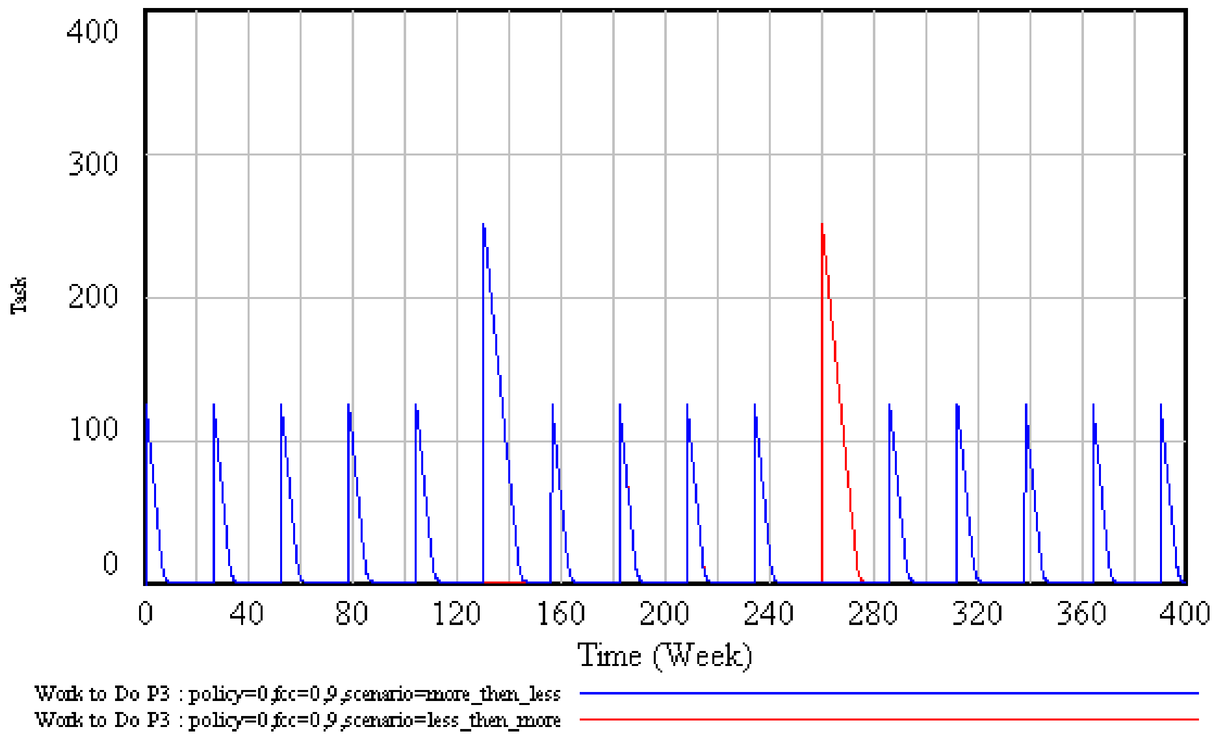


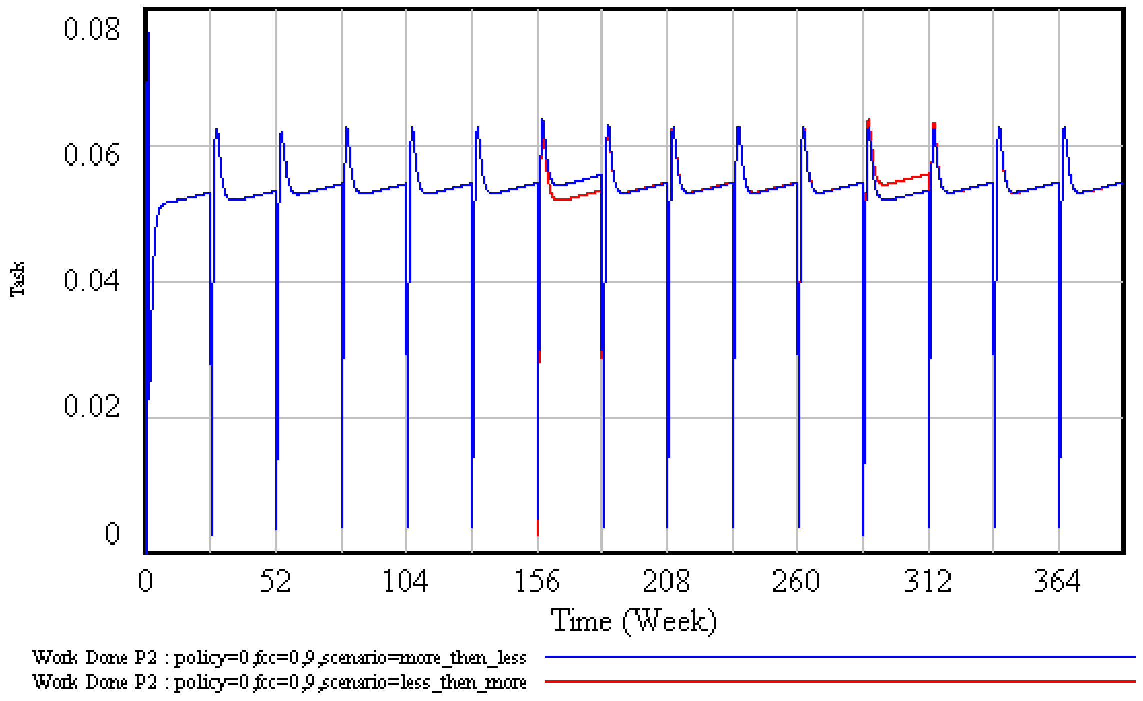
Figure 5-22: Shows how the input *Work To Do* is varied in the two experiments.

We run two scenarios:

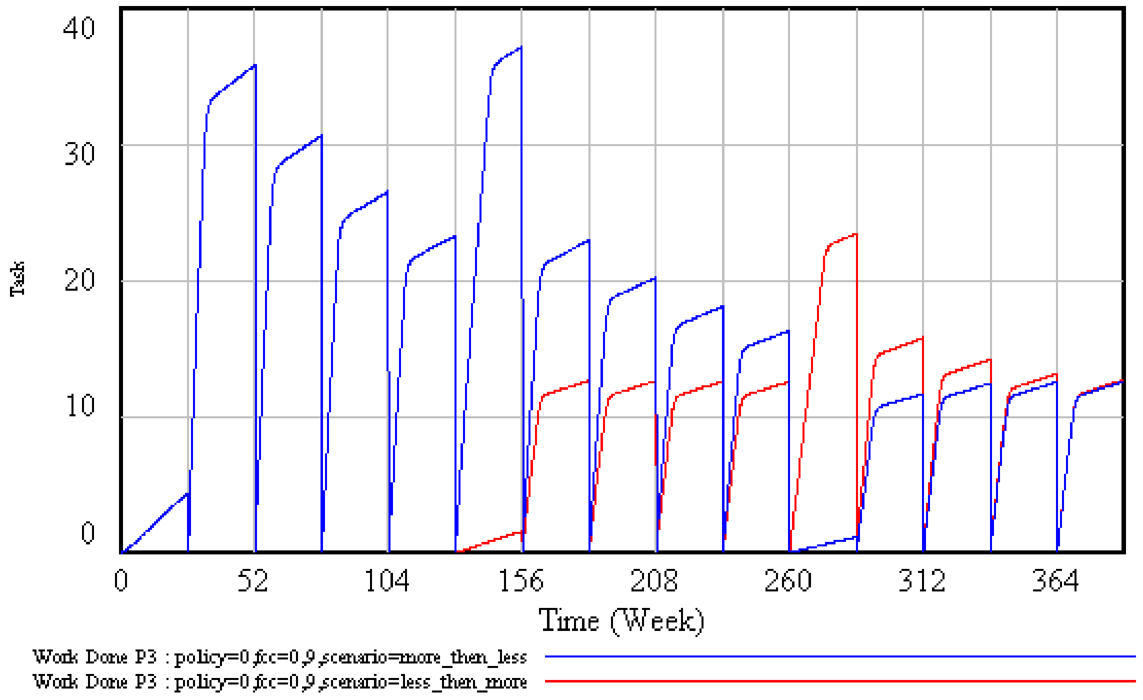
- The scenario depicted in red is a drop to zero in workload on release #5, followed by a doubling in workload on release #10. This is similar to the scenario tested by [Rahmandad & Weiss 2009], and replicates his results, which he explains as activating *virtuous cycles* that build up *organizational resilience* which allows the organization to survive the large shock of a sudden increase in workload.
- The scenario depicted in blue is the reverse: a doubling at release #5, followed by a drop at release #10.

These might appear to be extreme cases, but we use them for their probative value, not as a representation of reality.

The plots presented show the output results of the experiments, for *Work Done P2* and *Work Done P3*, as both stages present a variation in output.



(a) *Work Done* on stage P2



(b) *Work Done* on stage P3

Figure 5-23: Shows how *Work Done* behaves when propagation is active and we vary the workload on releases #5 and #10.

On Figure 5-23a, we can see that the variations are minor: this is because the development system is under capacity (by design, as indicated on page 105).

The non-linearity can be seen on Figure 5-23b, which shows that a drop followed by a spike causes the system to perform less work overall than the reverse. We could hypothesize that a sudden increase causes workers to work more intensely in the short term to get the work done, and that is accompanied by an increase in undiscovered rework and in the effort cumulatively expanded to accomplish all the tasks assigned. The sudden decrease scenario, on the other hand, allows the staff to catch up on *Work To Do* remaining in all stages of the system, which is beneficial only if there are pending tasks.

The managerial lever that this uncovers is that dropping the workload temporarily ahead of a planned increase seems more favourable to the intensity and quantity of work individuals will have to provide, we might say that it should tire the troops less. However, this is counter-intuitive: one might think that under a decreased workload, workers could get lazy and have a hard time getting their productivity up when workload increases again.

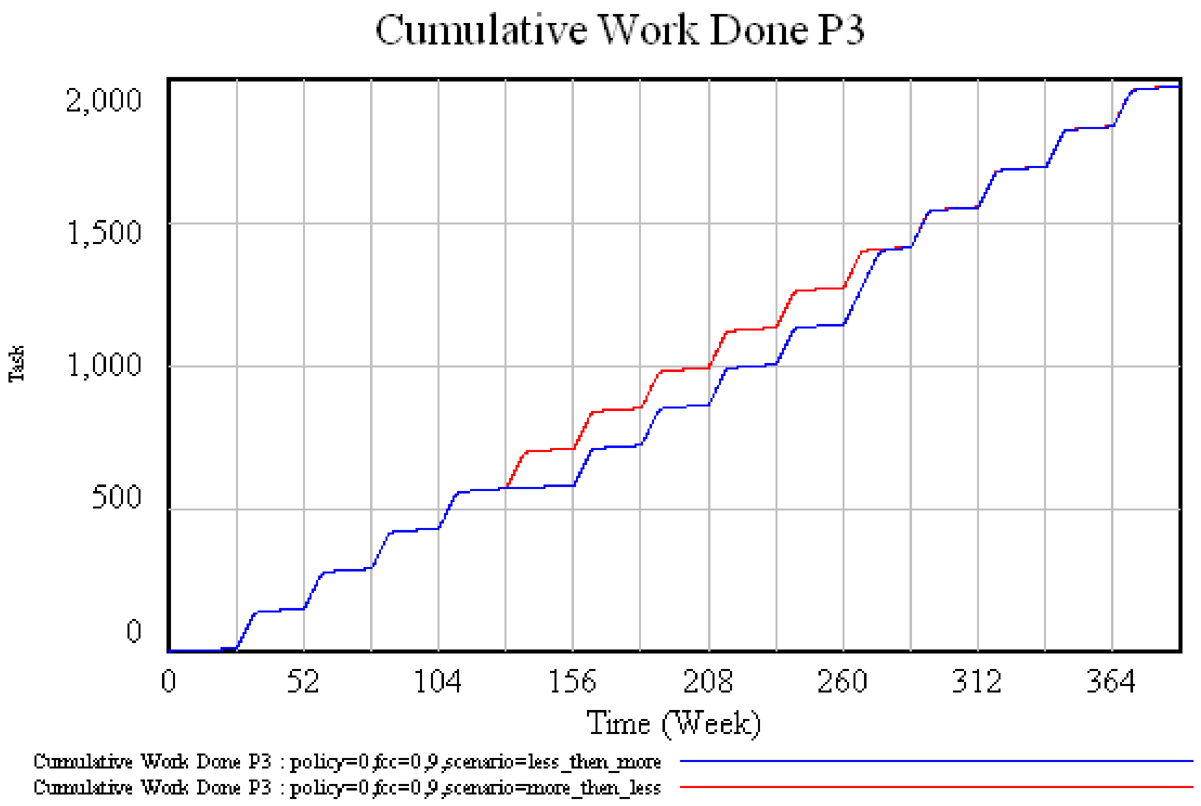


Figure 5-24: Shows how *Cumulative Work Done P3* behaves in the two scenarios.

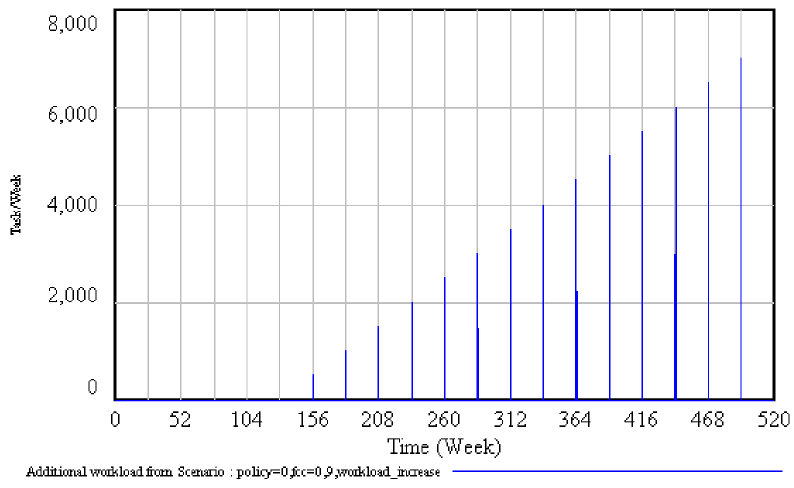
Comparing the peak at release #5 on the blue plot in Figure 5-23b with the peak at release #10 on the red plot, we can see that increasing-then-decreasing puts more strain on the production system than decreasing-then-increasing. This can also be seen on Figure 5-24, where increasing first clearly shows more resources spent for the same end result. The long-term effects are similar, as by release #15 both plots are the same. However the lesson for the engineering manager is that increasing-then-decreasing drains more resources, and is more work on the whole for the organization. In summary, managers should avoid sudden bursts in workload if they can, and if not, to preserve the troops, they should want to give them rest before to inflicting a burst in workload.

5.8.8 System Behavior under constant workload increase

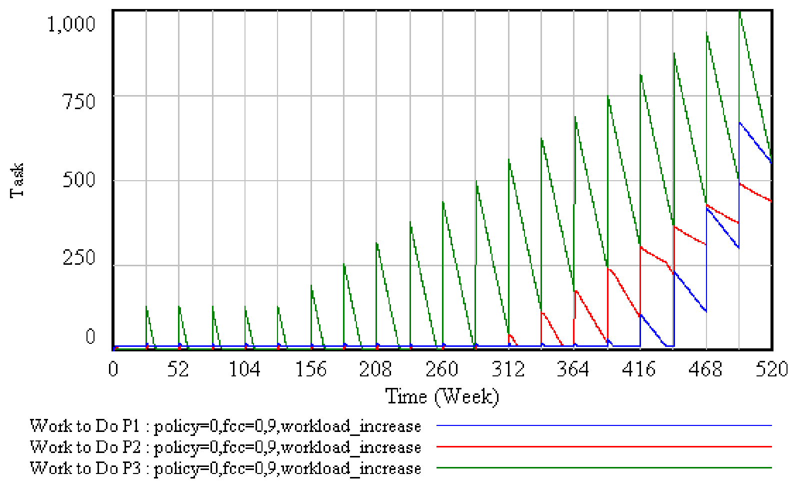
In this experiment, after letting the system reach its steady state at release #5, we subject it to a sequence of successive pulses, each increasing the workload at the start of a release cycle by 50% with a constant number of staff. The simulation runs for 20 years[1], that is 40 release cycles, to visualize long-term effects, as the case may be. We can see how the *Work To Do* trend increases linearly, while the *Work Done* reaches the maximum capacity (see Figure 5-25c) and so does *Rework Generation*.

Figure 5-25a shows the pulse train used as additional input in this experiment; Section 5.8.1.7 shows the machinery used in VENSIM to simulate this input. Figure 5-25b shows the corresponding output of *Work To Do* on all three stages, while Figure 5-25c shows the corresponding change in *Work Done*: we can see that *Work To Do* increases linearly without bounds, while *Work Done* reaches its maximum capacity at year 12. In the zone highlighted by 1 and 2, this system is capacity constrained, it works as much as it can but cannot produce more work than its maximal output capacity, and goes into a mode of *infinite defects*, which is what McConnell documented in [McConnell 1997, p. 204]: the system dynamics generate more defects than can be fixed, and staff only has capacity to work on the errors on errors, not on new *Work To Do*.

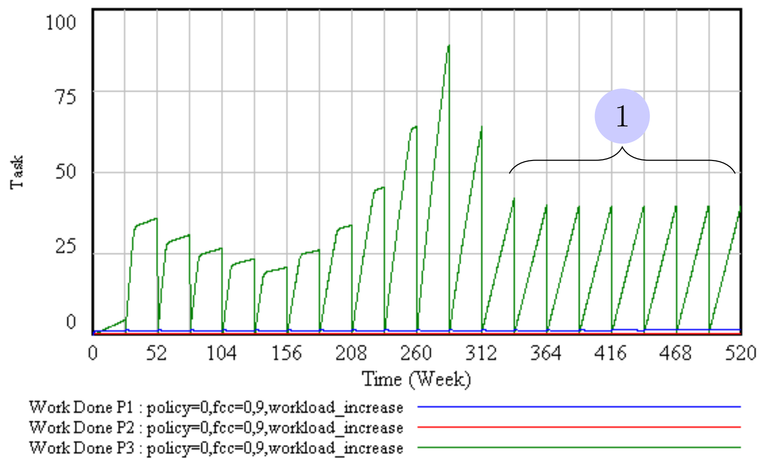
[1]admittedly a long time, but one can adjust for more specific circumstances.



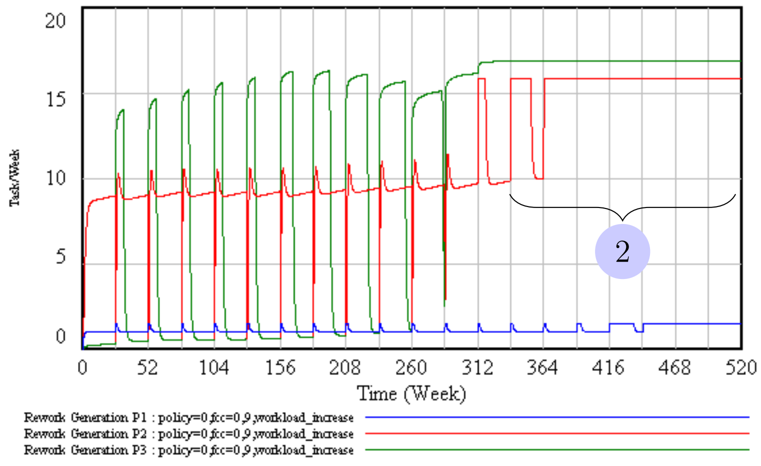
(a) Additional workload



(b) Work To Do



(c) Work Done



(d) Rework generation

Figure 5-25: Rework

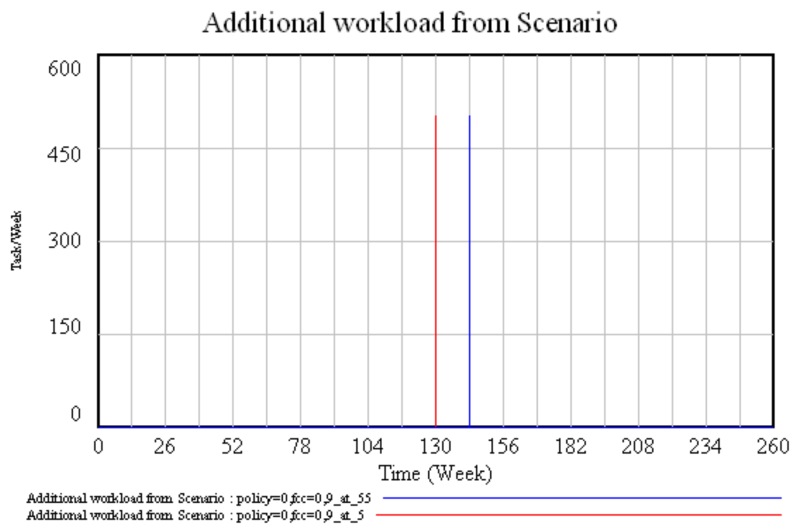
While purely synthetic, this simple experiment highlights the fact that the system has a fixed capacity and that managers ought to realize this when planning ahead for the number of tasks they intend to subject the system to.

5.8.9 System Behavior under workload increase late in the release cycle

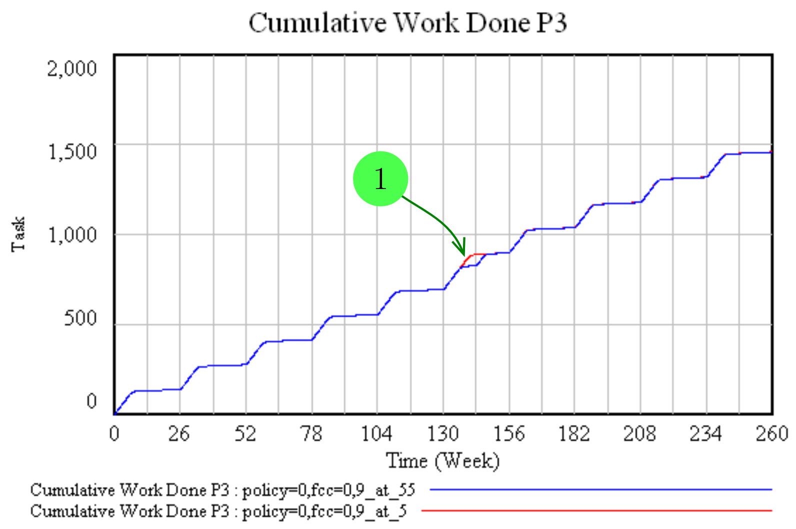
In this section, we experiment with the model and see what happens if we add a supplementary workload half-way through release #5. The intent is to simulate what happens if management commits to adding extra feature content while in flight.

Figure 5-26a shows the input changed for this simulation: we compare two bursts of extra work, one at release time (as in Section 5.8.5) in red, and another in the middle of a release cycle, shown in blue. *Cumulative Work Done P3* is shown on Figure 5-26b, the green arrow pinpoint a mere blimp in the quantity of work accomplished. Figure 5-26d shows that there is a small change in work performed (see 1), but the overall output is identical: the intuitive interpretation of the phenomenon shown here would be that adding work late in a release cycle makes no significant difference . One caveat we must add here is that this assumes that the work added late is independent of earlier work in the release and that there is no scheduling dependency between the two sets of tasks: an improved model could tackle this issue.

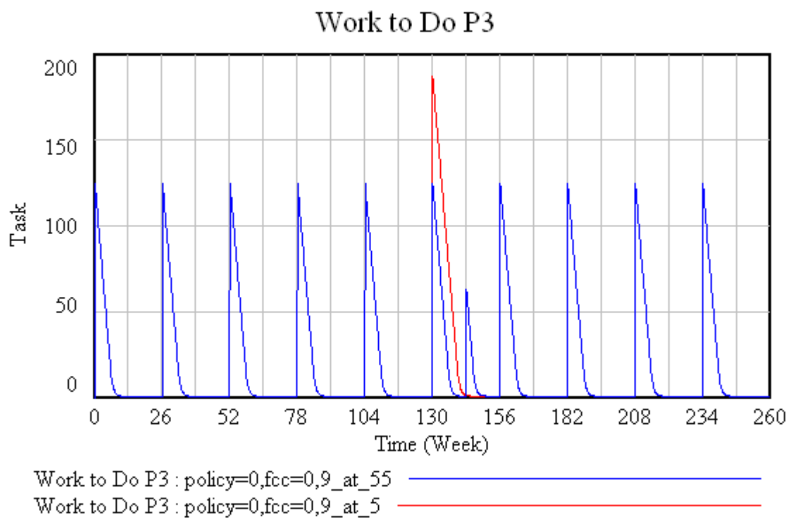
The management lesson here is that one may be able to add work late in a release cycle if there is spare capacity, which is difficult to estimate, and if the new set of features is independent of work done within this release.



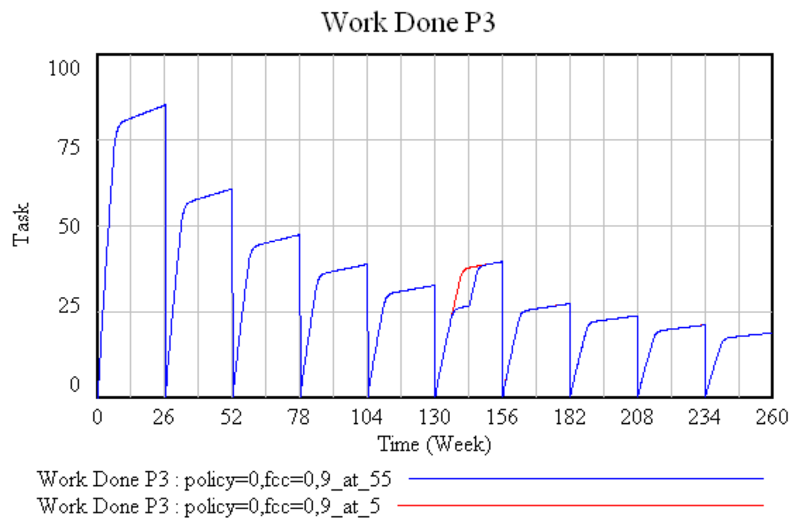
(a) Additional workload



(b) Cumulative Work Done P3



(c) Work To Do P3



(d) Work Done P3

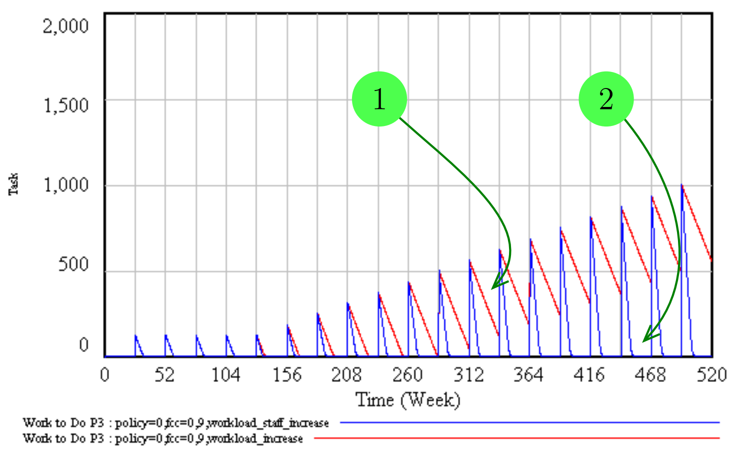
Figure 5-26: What happens when work is added mid-release.

5.8.10 System Behavior under simultaneous staff and workload increase

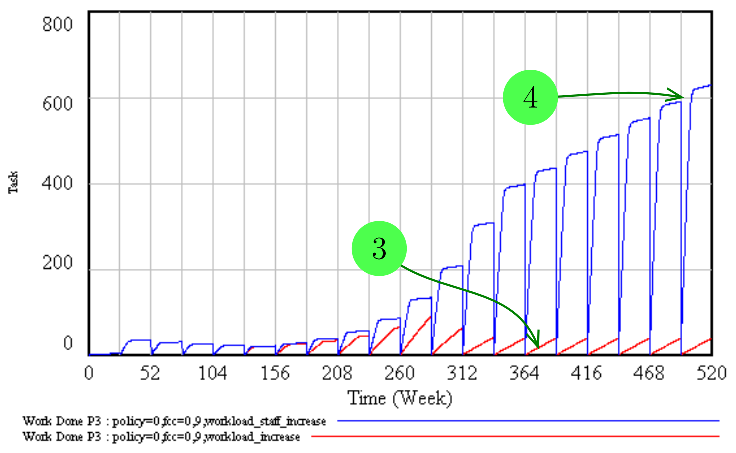
In the previous experiments of Section 5.8.5 and Section 5.8.8, we changed the workload while maintaining constant levels of staffing; If we now modify the previous experiment to also increase the staff in the same proportion as the workload, we get the plots of Figure 5-27. In red are the plots for the experiment run with only a workload increase. In blue are the plots for the experiment run with both workload and staff increasing in the same proportion.

Figure 5-27a shows the additional bursts of workload that we subject the system to. Figure 5-27b shows the corresponding change in work done, which clearly shows the system accomplishing more and more work as both tasks and staff are added. 1, 3 and 5 indicate the saturation of a system with too little staff. Figure 5-27c shows that the system (in blue) breaks out of the limitations imposed by a fixed staff (in red): in particular, 2 and 4 indicate that the output continues to grow as the system grows.

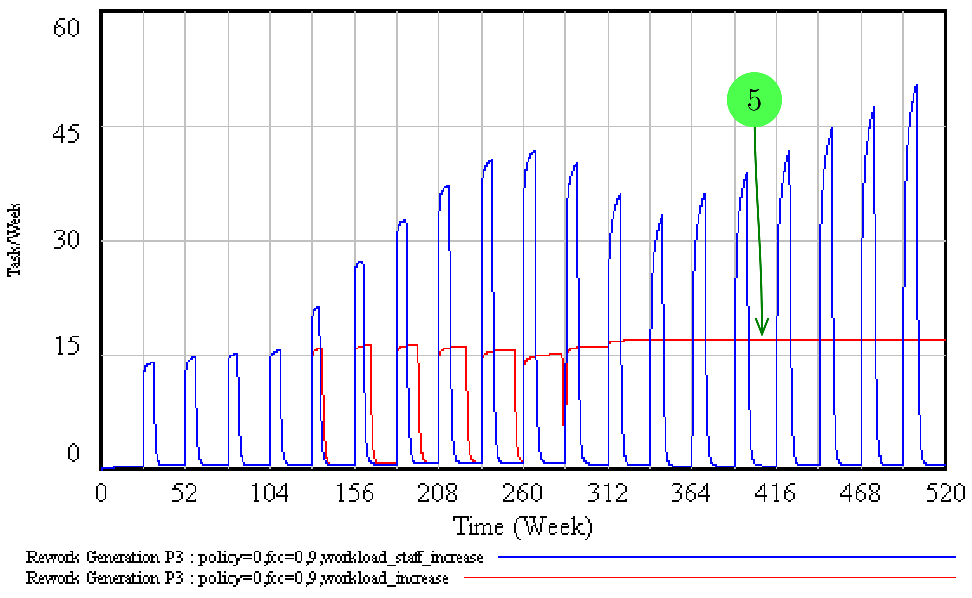
This experiment is at best an approximation since it does not account for individuals having to get experience on the particulars of the engineering of the specific product line. The rework generation exhibits much better (but still non-linear) behavior and the system as a whole responds better to the change in input. The lesson here is that the system can accomplish more with more people, but our model displays its limitation, in that it does not account for acquisition of experience by new hires. The manager should know that past some level of workload, increases in staffing are the only way to generate more output, but he should be aware of Brook's Law as in Section 3.1.1.



(a) Work To Do P3



(b) Work Done P3



(c) Rework Generation P3

Figure 5-27: Behavior as staff and workload both increase linearly.

A variant of the previous experiment consists in stepping up both workload and staff simultaneously: this simulates a merger between identically-sized two companies, or more precisely a merger between two product lines (which may happen for business reasons much later than the actual merger of companies, as seen in Section 4.2.2.2). We suppose the two companies go much further than a simple merger in the legal sense of the term: we suppose that they have decided to merge their separate operations into a single bigger operation. My industry experience is that merger Research & Development operations requires a lot of up-front preparation: the processes and build systems must be unified (which requires porting one of the two systems to the build system of the other), and then source control and defect tracking systems must be unified (source control history must be imported, simultaneously operation in two *SCM* systems maybe be temporarily necessary, etc...) to complete integration. We assume here that this work has all been performed done, and that release 5 is the time of the "big switch" to unify the whole software production system.

Consequently, here we propose an experiment where we double the workload and the staff at release 5.

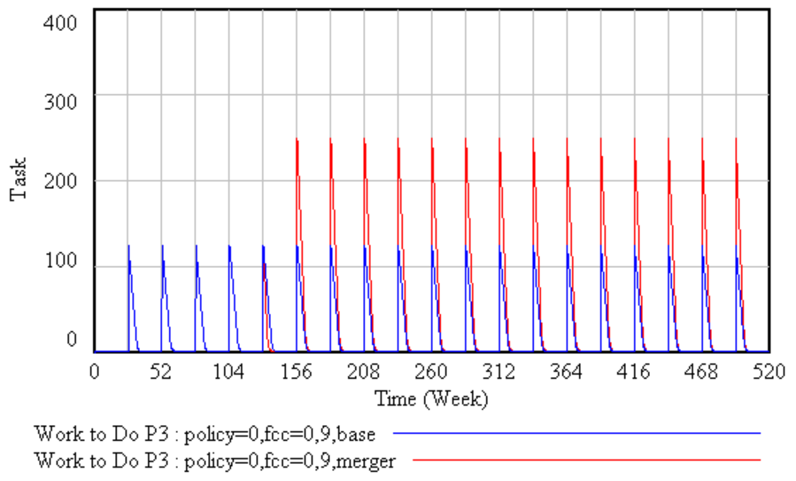
Figure 5-28a on page 141 shows the variation in workload: our baseline case is plotted in blue, while the merger case is plotted in red. The graph on Figure 5-28b shows the non-linearities: we observe that the *Work Done* on the merged system is more than simply the sum of the two *Work Done* quantities on two same-size systems. Another observation is that reaching actual capacity of an entity of equivalent size is not immediate, the system in fact takes a long time to adapt and converge. Figure 5-28c shows the convergence in terms of *rework generation*: the system transiently generates more rework than it finally will in its new steady state. The effects one stages P2 (shown on Figure 5-28d and P1) are not visible with this model, which is not detailed enough to display any insight on that aspect (especially considering the *other* system probably has its own P1/P2 stages that are maintained independently).

The point we want to make here is that, even under the best conditions, there is evidence of non-linearity in the merging of two software product development entities: the lesson for leaders is that one should not attempt such merges of development organizations lightly, as these represent a shock to the two pre-merger systems. Only when synergies are expected on the long-term should one attempt merging operations. When there is no strategic incentive to merge, it may be better to keep things separate.

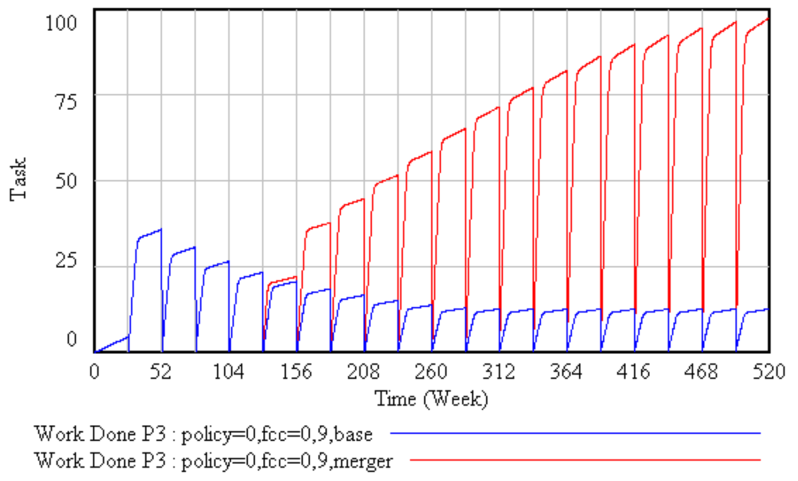
Another aspect to look at is the nature of the product one attempt to merge into the existing product family:

– if the product is situated in the *periphery*, then issues will be limited to the interface between that product and the rest of the software system. This is most favourable case (the one we could argue the previous experiment modeled).

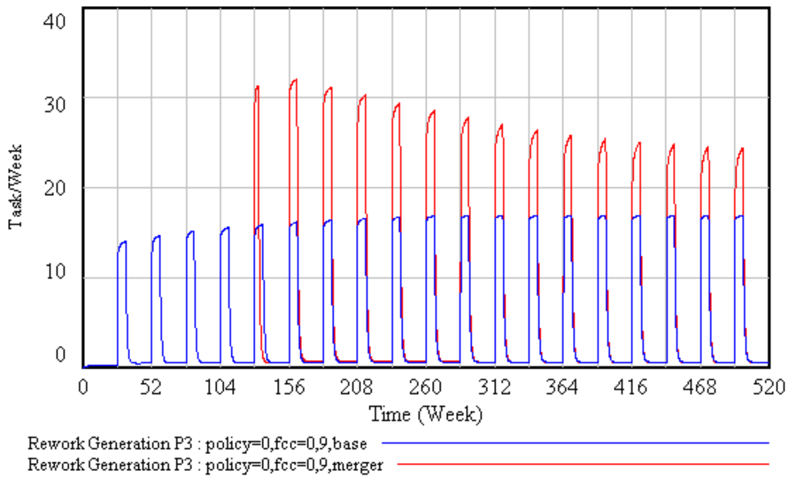
- if the product is situated in the *core* of the system, then we believe this experiment is not sufficient to reflect the reality of a merge. Complete integration requires a lot of care and might need to be performed much more gradually than the sudden event modeled here.



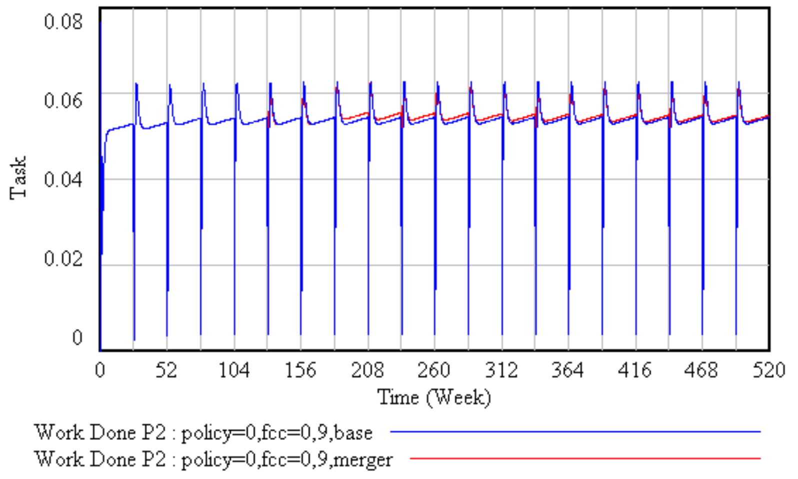
(a) Work To Do P3



(b) Work Done P3



(c) Rework Generation P3



(d) Work Done P2

Figure 5-28: Effect of same-size merger on software production system.

5.9 The Rework Cycle in the hierarchical delivery system

In a setting like that of Figure 3-12b, the software production system is a hierarchical delivery system. Each of the three stages in Section 5.8 in fact runs several nested rework cycles (as opposed to a single rework cycle in the model of Section 5.8). This is what we want to examine more closely in this section.

In this section, we take the stance that those *building blocks* mentionned in Section 5.4 can also represent both the stages of release cycles running in parallel, and within the *current* stage they can represent the various components and their associated organizational units, each with their own milestones and schedules. In the context of this thesis, different engineering disciplines, such as structural, electrical or power disciplines, in projects with a physical realization correspond to multiple organizational units all executing the software engineering discipline, but applied to a different component within the architectual design of the software system being produced.

As shown on Figure 5-29, the various components of the organization each have their own rework cycles (highlighted in red), each contributing feature content (rather than *Work To Do*) to the master rework cycle that represents the best-so-far version of the *current* release. When there is a dependency between an application team and the core team, that dependency is represented by the fact that the application team is waiting for the other team to delivery a minimum product in the best-so-far view of the *current* stage. Once this minimum delivery is published, the application team begins development and testing and finds issues that are reported to the core team: such is formed another cross-team rework cycle that spans the organization. Because this rework cycle spans the organization in a way that adds delays to the reporting of issues and their fixing, this subsystem will exhibit non-linear response when rework does happen. This is further worsened when work is distributed geographically, as shown on Figure 4-1. To reuse Sturtevant's wording [Sturtevant 2013], this is an *architecture spanning cycle*.

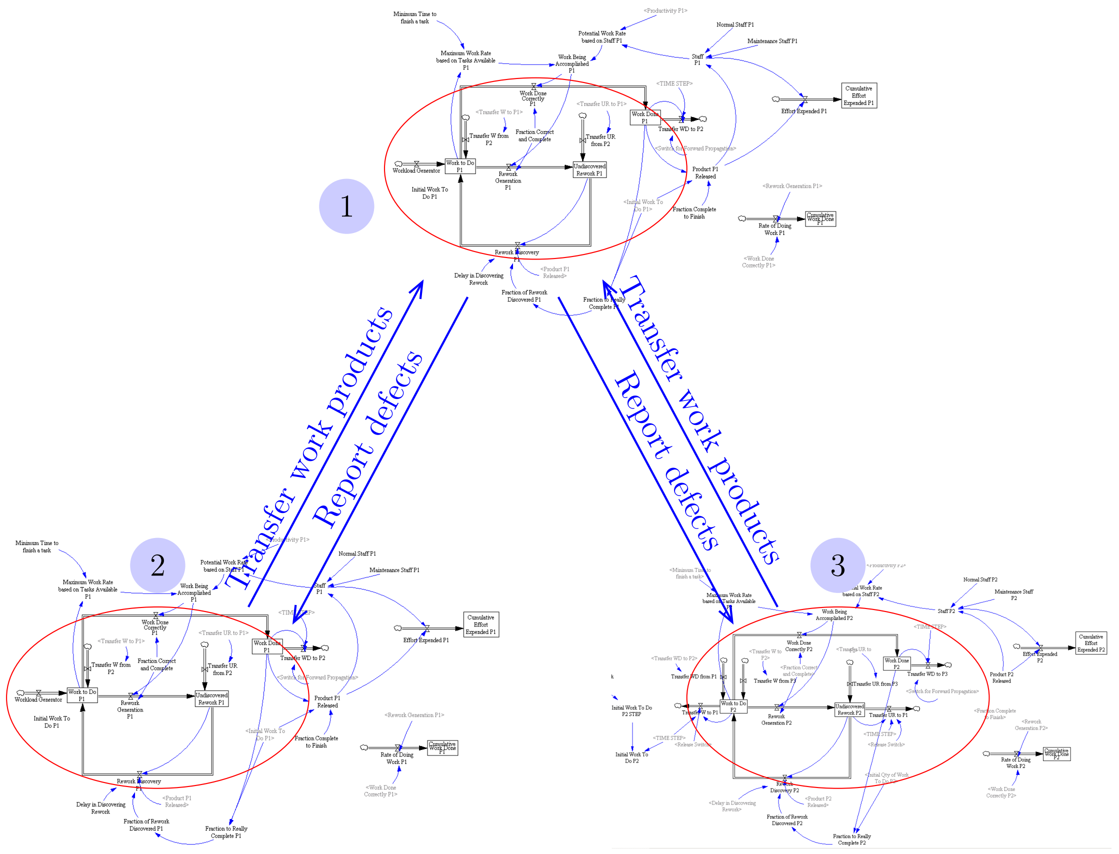


Figure 5-29: Example with one level of hierarchy and three rework cycles highlighted in red: 1 is the master repository, 2 is the "core" software production system, and 3 is the "application" software production system.

In a hierarchical delivery system, the timings of lifecycle of a component depends on that of the core components it depends on. In effect, a core component publishes a known-good or *Best So Far* version to its clients: this is very similar to fielding a release. What happens at the company level is that it fields a released product at the end of each period, customers then use this released version and file requests for enhancements that go into the next release cycle, as well as defect reports that they hope will be fixed in the current version. The core/periphery metaphor in effect reproduces the same schema at a lower level, inside the development organization: here the customer is the peripheral component, the provider is the core component, and fielding a new version is accomplished by publishing the new version to the rest of the organization. This change control policy enables concurrent engineering, at least for features that have no cross-component dependency.

[Abdel-Hamid & Madnick 1991, p. 79] and the literature considers losses due to faulty process referring basically to communication and motivation. This aggregate view is a simplification as loss of productivity also occurs when someone's mistake prevents others from working, or makes it more costly for them to re-setup to be able to work again.

In a hierarchical delivery system, we want to highlight that new personnel[2] has a higher propensity to make mistakes that cause rework and stalls. That reason and the fact that some core components are more complex [Sturtevant 2013] imply that people as units of staff are not all equivalent, and therefore a realistic model would have to account for this. This also implies that managers probably want their best developers to work on the core, and working on the more complex parts of the system is something one can do after a long period of training. Sturtevant's results show that there is increased difficulty in working on the core: these shine in the view of a hierarchical delivery model, as the core needs to have a slower and more cautious delivery cycle so as to guarantee the quality of its work product, and the stability of the overall software production system.

In conclusion, development in the periphery is development on top of a moving platform, therefore it is a requirement that the platform provide some promises of stability, so that product engineers operating in the periphery can perform their work without fear of disruption caused by platform changes. By using hierarchical decomposition in the change process of the various components and subsystems of the whole product system, one can manage tight iterations of rework at the level of each component so as to isolate other parts of the system from such rework iterations. This provides better scalability when many engineers work concurrently on various parts of the system.

[2]The model presented in previous sections does not account for hiring and firing of personnel.

5.10 Discussion of Results

Looking at completion time of each stage (i.e. when the stock of tasks drop to zero, accounting for the fact that defect discovery can occur long after the product has been fielded), and resources expended in each of the experiments we ran in this research project, we can summarize the key lessons learned:

5.10.1 Summary of lessons learned

- As observed in Section 5.8.3, in the situation of Software Product Line engineering with multiple active releases, the effects of rework propagated from *fielded* releases cannot be ignored. Project managers know that investing in front-end design activities produces a higher-quality work product; software product line managers should know that the effect is similar, but the consequences are dramatically worse when work is not done right the first time. The engineering manager should therefore invest in both design and validation activities on the *current* releases to decrease as much as possible the undiscovered rework that escapes to *beta* and *fielded* releases.

- As was shown in the experiments with staff allocation policies in Section 5.8.4, giving priority to maintenance work for *fielded* releases can result in firefighting that consumes all the resources of the development group. The engineering manager should exercise policies that avoids firefighting: this can be done by limiting the staff doing maintenance work, and more preventatively by investing in front-end activities to limit the undiscovered defects and rework that escape in the field.

- Section 5.8.5 showed that sudden variations in workload increases can have long-term effects. It seems that there are two aspects to this. First, the successful software development company wants to grow: the lesson here is that growth should be gradual rather than sudden. Second, the engineering manager faces pressure from customers to always include their favorite feature in the *current* release: the lesson from that standpoint is to spread development work over many release cycles. Prioritization and scheduling are the tools the manager can use to maintain a manageable scope of features in each release.

- The experiments in Section 5.8.6 and Section 5.8.7 seem to show that a temporary decrease in workload lets the organization recover from previous shocks. [Rahmandad & Weiss 2009] explained this by saying that such periods of lesser workload must be *virtuous cycles* that build up the *organizational resilience* and the capabilities needed to support a later increased workload. A more sceptical view of the experiment would argue that there is a risk that laziness and complacency would set in and that a future increase workload would become even more difficult to support.

- In Section 5.8.10, we ran one experiment that show saturation of the production system when it is under-staffed. The lesson is that the manager should grow his staff as the size of the product family increases, the difficulty, of course, is to know when to hire. The SD model we used in this work does not account for hiring and firing, nor does it account for "learning curve" effects: these two phenomena happen in long-lived enterprises, therefore an integrative model including those aspects would surely be more representative of real situations. In the same Section 5.8.10, we simulated crudely what happens during a merger of two equal-sized software development organizations. This synthetic experiment may not represent a real situation, but yielded the lesson that even under the most favorable conditions, the system behavior is non-linear. A more interesting model would integrate the notions exposed in Section 5.9 and the specifics of a merger: for instance, observing differences in system behavior when the merger target is a smaller entity engineering products either in the *core* or in the *periphery* would be of interest. The intuition is that acquiring and merging a new product member in the *periphery* is easier and inherently less risky: the product line development manager would surely like to know ahead of time what to plan for in such a situation.

- It might be obvious in retrospect, but the workload created by *fielded* releases of a software product system can require significant maintenance expenditures from the company. The modeling presented here collapsed all *fielded* releases into one stage, but we can posit that the workload for that one stage is proportional to the number of releases active in the field. Since the maintenance workload is correlated with the number of active releases, it seems that keeping that number low is very desirable. Having too many active releases is one way the staff can spend a lot of time task-switching and otherwise performing work that is subject to overhead. Figure 5-18a illustrates that if the quantity of work generated by fielded releases rises and the staff allocation policy favors *fielded* releases systematically, then there will be a point when the organization as a whole does nothing but firefighting on old releases. Therefore, to remain lean, the organization should want to limit the number of active releases, as this will avoid dispersion of staff and will curtail propagation of rework across releases. This policy should also limit firefighting. The ultimate version of this is to have a single active release, which is what companies, like Amazon® or Google®, have with their continuously-released hosted applications: instead of having the extreme one-version-of-the-product-per customer syndrome, these have one-version-for-all-customers.

5.10.2 Discussion of lessons learned

In light of the key lessons learned listed in the previous section, some points can be further discussed as they relate to the problems mentioned before:

- Following [Wheelwright & Clark 1994, p. 152], one must ask the question "*is the development system overloaded?*" and discover the actual capacity of the software production system. This can be inferred approximately from the analysis of historical data relating code size, number of defects and time to completion. We have seen that overloading the system causes long-term negative consequences. It follows that any growth must be incremental and tightly monitored. Another conclusion is that the product line engineering managers should assign a workload below the actual capacity of the production system at their disposal, to account for the time and resources expanded on work caused by previous active mainlines. Even if the real workload generated by parallel mainlines is not predictible, the manager can and should keep a reserve in resource expenditures to manage that workload when it happens.

- The negative consequences of overloading the software production system above its intrinsic capacity are devastatingly more important than the positive consequences of decreasing its workload. Managers should strive to prevent exceedingly bad outcomes by fitting the workload to the system.

- Software product lines in the context of periodic release cycle present a unique opportunity for organizational learning, as the software production system becomes optimized for the production of its particular software products. This includes deploying and adapting its resources to sustain its throughput; in other words, this is about:

  - knowing to spend time and resources at the front-end of development, by investing in careful design and implementation, as this increases quality and therefore decreases future rework,
  - knowing how many parallel active mainlines to have,
  - knowing how to best use its people,
  - knowing to avoid increasing the workload even "temporarily" as effects last longer than expected
  - knowing how to spread the workload over many successive release cycles.

  Because the time horizon is much longer than that of a single project, and instead of being victim to the pernicious effects of cross-mainlines rework, the manager can and should take advantage of the situation to achieve repeatability and predictability of outcomes of the software production system.

- When a low quality product is released to the field and numerous defects come back to the software development organization, it is clear that a really poor level of quality can hurt the development of new features, to the point of stopping it entirely. Devoting some resources to maintenance is necessary to make sure the products are useable in the field, this is indispensable to keep customers happy. A second point is that lowering temporarily the workload of the development team allows the organization to eliminate some of its backlog of defects, affecting positively the quality of the products. Nevertheless, managers cannot put staff on maintenance-only tasks for too long. This leads us to posit that engineering managers should try to conjugate acceptable levels of initial quality (to keep the maintenance workload at a low enough level) with periods of lesser development activity.

Another way to put it would be to say that the development organization has a finite capacity that cannot reasonably exceeded sustainably for too long, and alternating between feature-loaded release cycles and maintenance-only release cycles would seem to be a good way for the organization to breathe (just like individuals, it needs periodic vacation!). One possible policy could be for the organization to have a rotational program where some teams are in feature-development mode, while others are in defect-fixing mode. Rotating staff is difficult in the model presented here, because all staff is seen in aggregate: this aspect of modeling is too coarse, we would need a finer notion of staff associated to a particular stock of work (perhaps one related to work in one particular component, to match the details of hiearchical domain engineering organizations).

- Labor can be divided along many dimensions:
  - TIME-STAFF division: the engineering manager may decide whether to use dedicated staff for maintenance of *fielded* releases, to counter the effects of sharing staff across many active mainlines.
  - TIME-SPACE division: another possibility is to split the work among locations, as seen in Section 4.1.3, and let a remote entity work on the maintenance of *fielded* releases, or maybe some subset of those.
  - SPACE-PRODUCT division: distributing software development work across locations the work on various parts of the product line, for instance dedicating a team on a remote site to work on a product which is a leaf in the dependency graph of the software product line architecture.
  - TIME-PRODUCT division: schedule work on components such that there be only one moving part in a given release cycle, for instance, select component A to have feature content in the current release cycle, but let component B not have any. Instead of putting pressure on all parts of the software production system simultaneously, this rotates the pressure period around. It can also be used to build capability within each development group: the breathing period can be used to catch up on late defect fixes, to invest in design work for the next set of features, to invest in code refactorings that increase the value of shared assets, to bring new personnel on board, or to bring remote-location personnel on location

temporarily to build capability (for instance to delegate future maintenance work to that remotely-located staff). Note also that in the presence of shared assets, there is often a natural order of work: the work on the shared assets must be performed first, before a product engineering team can use the corresponding work products. This means that the product engineering team needs to wait for some feature to become available. In this division scheme, the engineering manager must be careful not to create long-delay cycles of rework that cross geographical boundaries, as that builds fixed minimum delays into the software production system feedback loops.

Staff and Space are necessarily strongly coupled, and while a practice often seen in industry is to assign product ownership with a specific group regardless (a staff-product division of labor), engineering leaders can use the other options for dividing work so as to maximize usage of resources and to build the capabilities their organization needs to continue producing larger software systems.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Topics of future study

As any time-boxed project, this thesis could only ever hope to include a limited amount of material, but over the course of the work, a few potentially interesting avenues of research were elicited: this is what this chapter provides.

6.1 Proposed Extensions to the Model

Like many models of complex systems, the model presented here is at best an approximation of reality. Yet it would be interesting to provide means for an enterprise to calibrate the models with historical data to compare past trends they have noticed and decide whether the model fits their reality or not. Then the model could be used not to predict outcomes, but to help understand these trends, determine project feasibility and pinpoint the unfeasible paths through generation and analysis of *what-if* scenarios. Such a model can be calibrated to reflect the particulars of one organization, and could be used in this setting to understand trends as well as effects of policies selected by the managers. Unfortunately, we have not been able to do this within the course of this research.

We propose here a set of possible extensions of the present work:

- In software product lines, the management of a portfolio of applications is another topic that includes the addition and removal of applications in the portfolio, or more generally of components available to build applications. This portfolio management aspect was not covered in the present thesis, but would be an interesting avenue of research. This can have effects on both the capability and capacity to produce the software products. An interesting issue is using development capacity to make portfolio management decisions: for instance, which applications to discontinue, which features to postpone (for instance to invest for future benefits in shared assets at an immediate cost in time-to-market for one particular product, as highlighted by [Bosch

2000, p. 321]), etc... to keep the software production system operating within capacity, and if possible near capacity, in order to maximize the best use of staff.

- One may want to link this to studies to compute defect densities with good confidence, thereby being able to put a dollar value on the maintenance cost . Managers would like to see Net Present Value calculations in place to be able to decide affordability of some system changes. For instance, is incorporating an acquired entity into the system feasible? What would be the cost?

- Similarly, refactoring can be viewed as a non-functional feature of a given mainline. Everybody on the engineering staff usually agrees that some of these refactorings should be performed (See the concept of *Technical Debt* by McCormack, also cited by Sturtevant [Sturtevant 2013]), yet these rarely get the visibility and priority required and therefore do not get done, leading to further erosion of the product architecture. Ideally one would like to be able to see the effects on the system of adding a step to the input stock of such work. This can help the engineer leader in making decisions to prioritize and schedule *Technical Debt*-related tasks: in particular, this kind of work touches existing function, so must be regression-free, while new feature development work should not impact the existing software system. In other words, scheduling refactoring work may be considered more costly than new feature development work: we might like to see historical data collected on this aspect, so that this can be factored into the system modeling.

- Wheelwright & Clark [Wheelwright & Clark 1994, p. 147] noted that *digging into the issues of portfolio management and capacity matching requires extensive information about resources and resources availability.* Similarly [Wheelwright & Clark 1994, p. 157], *rigorous, systematic, and objective analysis of development proposals* must be used to do portfolio management. A model such as that presented here can help use analytical thought to do selection of products-to-develop, rather than let gut-feeling or politics doing the choosing. Further study is also needed in designing methods to determine capacity of the development organization. One common industry practice is to pad schedule with "management reserve" to account for the *firefighting* and *rework* phenomenon, but one wishes analytical way of determining capacity were available. As it is now, the manager can know whether the production system is operating under capacity or over capacity, but sizing capacity is not a scientific process.

- This study only looked at *Variability in time* of a product whose space is fixed, further studies should look at what happens when *Variability in space* is added. In particular, the ability to run what-if scenarios for re-architecture should help managers see the benefits, if any, of rearchitecture.

- For the present paper, we simplified the modeling in VENSIM®, such that no stochastic modeling was in fact used. A more advanced model would describe failures, rework discovery, stalls in a stochastic fashion. This could be calibrated from historical data to match as closely as possible an organization (the larger the organizations, the more accurate its observations in aggregate on the propensity of engineers to make errors that cause failures, rework or stalls). In particular, the model considers fielded releases in aggregate, and one might want to investigate improving the model to better fit reality by having distinct Rework cycles for each fielded release, each with a workload generator that fits historical trends.

- An interesting extension of the model presented here would account for the probabilistic distribution of errors and of their importance. Since the software engineering stage feeds into the quality assurance stage, and the engineers working in the first stage sometimes make mistakes (of the kind introduced in Chapter 2) that make it impossible for anyone in the second stage to perform any work: this is a situation managers want to avoid at all costs, as employment of quality assurance staff is a sunk cost. The idea here is that occurrences and lengths of stalls would be modeled in a more accurate fashion. One would then use Monte Carlo simulation to run many trials and build confidence in results. In industry, it is important that the manager organize work in such a way that there is always a testable product for the quality assurance stage, so that QA staff can at least avoid being barred from working, and try to produce useful output (perhaps testing some other part of the system), therefore one would like to pinpoint the policy changes that improve this situation. For instance, we could conjecture that inserting a delay would improve the situation by isolating work in the two stages.

  We may note here that in industry, companies often keep a last-known-good build on hand, so that testers and other stakeholders can use that if the daily build is broken. This can be used in the short-term to keep the testing stage busy, but as noted in Section 2.4, the organization cannot let the daily build remain broken too long, otherwise there is a danger that testers will perform irrelevant work. The model would have to account for that in its delays.

- Another extension of the model presented here would not rely on a single aggregate view of *fielded* releases, and instead of using aggregated model of those, a more detailed version of the model would account for the exactly number of active releases, and the expected defect density on each of them to represent the realities of one company's software production system more closely. In particular, this would give a much more precise idea of the trends as the number of active releases increases, and their timing differs. When supporting real customers, the organization will ship out incremental fixes, sometimes called *service packs* or *hotfixes* or *patches* or *engineering builds*. The timing of those may not be on a fixed schedule, it may be modified at the customer's request. Another aspect is that delays in discovering rework may vary for this stage: as the software producers need to exercise more care when shipping incremental fixes for customers to use immediately in a production environment, they will intensify their testing to avoid causing regressions for customers. This is another level of modeling that could be done within the framework which was proposed here.

- One limitation of the model demonstrated here is that the assumption that schedule pressure functions exactly the same way in the framework of successive product line releases as it does in a single time-boxed project. This does not model reality perfectly: although there is some pressure towards the end of a release cycle to get as many as the planned features into the releasable product, the organization has a lot more freedom than finite-time projects do, the option of postponing completion of work into the next release cycle is present. Another way to think about this problem is that the schedule is on the many-releases master planning level, not on the level of planning each individual release cycle. In particular for technical features, which are not visible to end-users, but are visible to other product engineering units within the organization, the pressure comes from those units having to attain their own schedule objectives, which may be several release cycles away. This is part of a larger modeling effort that would account for the planning process as well as the long-term product and portfolio management process.
- An interesting aspect warranting further experimentation and analysis would be to simulate what happens when upgrading a third-party component: this event can be modeled as the addition of supplementary undiscovered rework. Let us assume that no extra work needs to be performed to integrate this new version of the same component: that is, we assume that the API of the component is stable: this would happen for instance when the newly released version from the third-party vendor is merely a bugfix release. One way to interpret this in our model is to suppose this corresponds to the mere addition of some quantity of undiscovered rework in the release cycle:
  - undiscovered rework in the third-party component, that is latent defects in the changed portions of that piece of software
  - as well as undiscovered rework in the organization's own software, as it could have previously and unknowningly relied on a bug in the previous version of the component, or have some code that used to function but is not allowed anymore in the new version. This happens typically when the third-party component tries to enforce tighter conditions on its inputs.

  In summary, upgrading third-party components causes perturbations in the system, and those would be an interesting topic for further research.

6.2 The push for continuous delivery

In the internet age, the delivery model of software applications has changed from local installation by the end user on their computer to instant delivery over the web. In effect, the web browser has become a platform that can replace the end-user operating system platform: developers can write their software using any combination of web technologies. The success of such technologies is due in part to their ease of deployment: in fact, the end-user does not participate in the deployment phase at all: web applications can be engineered so that upgrades to new versions can be done apparently on the fly while the system is online, without end-users noticing. For instance, large web sites switch to a newer version of their software without apparent downtime. This requires the operational capability to deploy to a fraction of users before gradually deploying to all users. This is usually done by partitioning users by geographical region or by partitioning servers in slabs, then doing multiple successive deployments, possibly stopping and reverting if a problem is uncovered. Even if a company has such a capability, the software applications must be engineered to support this mode of deployment: strict upwards compatibility must be observed at all times.

Customers of large software systems now expect web front-ends for these systems, if only to perform some simplified functions without requiring an explicit deployment phase. In the enterprise software world, this decreases Total Cost of Ownership by eliminating the need for enterprise-wide deployment, which used to be performed by the IT department. The second phenomenon is the emergence of hosted/cloud-based software.

In light of this, end-users and customers want reduced *Total Cost of Ownership* and may be willing to use a hosted service, thereby outsourcing the IT functions of production operations of the system to the hosting company, which may or may not be part of the company producing the software, and in parallel, producers of software want reduced maintenance costs, which they can obtain by reducing drastically the number of variants of their product in use at any time. It maybe even be possible to reduce it to a single variant, as opposed to many maintenance-only branches in the old world of big-bang shrink-wrap delivery.

This presents tremendous opportunity for software producers, as it can reduce their cost of ownership and maintenance of their software product line, while at the same time allowing them to lock their customers in by hosting their data. This simplifies maintenance by reducing the number of active branches in production, and simplifies online operations, as the data is really on-site with the producer, simplifying debugging, as there is no need to exchange data between the person who found a defect and the actual engineer assigned to debugging it: the engineer can directly request a copy of the data from the hosting operations staff. This also presents great risk, as moving from the old model to this new continuous model of delivery is a change in pace of delivery, towards a new pace that the delivery system has never been tested with. In particular, continuous delivery requires the ability to upgrade parts of the running system, progressively to full deployment, and to roll-back instantly a new version of the software: if the organization does not develop this capability, it is far too easy to make the product unusable and effectively be unable to provide the service while engineers are struggling to debug the issue live.

In a continous delivery environment, the system must always be working and functional, which is a characteristic systems described in this thesis do have. Build breaks cannot be tolerated, neither can regressions in tests. In fact, one may observe that continuous delivery systems have requirements in terms of availability that are quite close to those of a very large code production system where no single engineer can reasonably build the whole system, and thus there are constraints of availability much like those detailed in chapter 3. This shift in speed of delivery represents an increase in clockspeed of the industry, as defined by [Fine 1999], but also will highlight the tension between the “agile” style of development and the size of features visible to end-users: some features will be too large to fit in an “agile” *sprint*, requiring further chunking and planning of work. Significant modeling effort would be required to represent these changes in constraints, that would indeed be an interesting topic of study related to current developments in the software industry.

Chapter 7

Conclusion

In this thesis work, we followed the approach recommended by Wheelwright & Clark [Wheelwright & Clark 1994, p. 157] when they noted that *management exercises its greatest leverage by working on patterns and their causes*, attempting to gain insight into the long-term software development discipline for product lines. We have shown possible ways for software development managers to partition work such that the throughput of the development organization is maximized. The model presented is necessarily approximative, but it was developed for the explanatory value behind our theory, knowing fully the limits of the model. Hierarchy and modularity are the two basic ways of partitionning that engineering disciplines use, and indeed the same principles can be adapted to division of labor, and we have shown that these provide the same effects of stability in how project work unfolds. In other words, one outcome of this research is that designing the software production system is coupled with the actual architecture of the product line, each influences the other. This is one thing engineering managers should be aware of.

In addition, we have seen that the basic intuition that firefighting has devastating effects is correct, and that therefore organizations should minimize it using Repenning's recommandations. Another recommandation we can add here is that limiting the number of simultaneously active branches of development is a powerful tool to counteract firefighting.

[Wheelwright & Clark 1994] remarks that

> *Effective senior managers recognize that their most important contribution is their cumulative impact, rather than their influence on any single project. They act on the development process as a whole.*

In many ways, the issues that software development organizations face in product development really are centered around the chaotic situation individuals are in in the absence of clear direction and management. By imposing process and structure on the way people divide and perform labor, there is tremendous opportunity for the leaders to increase the ratio of useful work to unnecessary work: using an information theoretic metaphor, one could say the manager wants to reduce the entropy of the organization.

We hope that this thesis has presented some insights of interest to the practitionner, and thus should help *make good cultural practices transferable* [Weinberg 1991, p. 294].

Bibliography

[Abdel-Hamid & Madnick 1989] Tarek K. Abdel-Hamid, Stuart E. Madnick, *Lessons learned from Modeling the Dynamics of Software Development*, Communications of the ACM; Vol 32, No 12, pp. 1426-1436, December 1989.

[Abdel-Hamid & Madnick 1991] Tarek K. Abdel-Hamid, Stuart E. Madnick, *Software Project Dynamics: An Integrated Approach*, Prentice Hall; April 29, 1991. ISBN-13: 978-0138220402

[Acuña, Juristo, Moreno & Mon 2005] Silvia T. Acuña, Natalia Juristo, Ana María Moreno, Alicia Mon, *A Software Process Model Handbook for Incorporating People's Capabilities*, Springer; June 28, 2005. ISBN-13: 978-0387244327

[Akaikine & MacCormack 2010] Akaikine A., MacCormack A. D. (advisor), *The impact of software design structure on product maintenance costs and measurement of economic benefits of product redesign*, S.M. Engineering and Management Master's Thesis, System Design & Management Program, Massachusetts Institute of Technology, 2010.

[Allen 1984] Thomas J. Allen, *Managing the Flow of Technology: Technology Transfer and the Dissemination of Technological Information Within the R&D Organization*, The MIT Press; January 4, 1984. ISBN-13: 978-0262510271

[Berczuk 2002] Stephen P. Berczuk, Brad Appleton, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*, Addison-Wesley; November 04, 2002. ISBN : 0-201-74117-2

[Boehm 1981] Barry W. Boehm, *Software Engineering Economics*, Prentice Hall; November 1, 1981. ISBN-13: 978-0138221225

[Boehm 1986] Barry Boehm, *A Spiral Model of Software Development and Enhancement*, ACM SIGSOFT Software Engineering Notes; Vol 11 (4), pp 14-24, 1986.

[Bosch 2000] Jan Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach* Addison-Wesley Professional; May 29, 2000. ISBN-13: 978-0201674941

[Box 1998] Don Box, *Essential COM* Addison-Wesley Professional; January 1, 1998. ISBN-13: 978-0201634464

[Brooks 1975] Frederick Brooks, *The Mythical Man-Month*, ADDISON-WESLEY; 1975. ISBN-13: 978-0201835953

[Conway 1968] Melvin E. Conway, *How do Committees Invent?*, DATAMATION; Vol 14 (5), pp 28-31, April, 1968.

[Cooper 1980] K. G. Cooper, *Naval Ship Production: A Claim Settled and a Framework Built*, INTERFACES; Vol 10 (6). 1980.

[Cooper 1993] K. G. Cooper, *Swords & plowshares: the rework cycles of defense and commercial software development projects*, AMERICAN PROGRAMMER; Vol 6 (5), p 4151.

[Cusumano 2004] Michael A. Cusumano, *The Business of Software: What Every Manager, Programmer, and Entrepreneur Must Know to Thrive and Survive in Good Times and Bad* FREE PRESS; March 2, 2004. ISBN-13: 978-0743215800

[Cusumano & Selby 1998] Michael A. Cusumano, Richard W. Selby, *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets and Manages People* FREE PRESS; December 4, 1998. ISBN-13: 978-0684855318

[Cusumano & Selby 1997] Michael A. Cusumano, Richard W. Selby, *How Microsoft builds software*, COMMUNICATIONS OF THE ACM; Volume 40 Issue 6, pp 53-61. June 1997

[Dikel, Kane & Wilson 2001] David M. Dikel, David Kane, James R. Wilson, *Software Architecture: Organizational Principles and Patterns*, PRENTICE HALL; January 7, 2001. ISBN-13: 978-0130290328

[Fine 1999] Charles H. Fine, *Clockspeed : Winning Industry Control in the Age of Temporary Advantage*, BASIC BOOKS; October 1, 1999. ISBN-13: 978-0738201535

[Gilb & Graham 1994] Tom Gilb & Dorothy Graham, *Software Inspection* ADDISON-WESLEY PROFESSIONAL; January 10, 1994. ISBN-13: 978-0201631814

[Glaiel 2012] Firas Glaiel, *Agile Project Dynamics : A Strategic Project Management Approach to the Study of Large-Scale Software Development using System Dynamics* MIT SDM Thesis, May 2012.

[Goldratt 1997] Eliyahu M. Goldratt, *Critical Chain*, THE NORTH RIVER PRESS; April 1997. ISBN-13: 978-0884271536

[Gorton 2011] Ian Gorton, *Essential Software Architecture* SPRINGER; May 5, 2011. ISBN-13: 978-3642191756

[Grady & Caswell 1987] Robert B. Grady, Deborah L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, PRENTICE HALL; June 6, 1987. ISBN-13: 978-0138218447

[Gray 1985] Jim Gray, *Why Do Computers Stop And What Can Be Done About It?* TANDEM COMPUTERS; Technical Report 85.7. 1985.

[Humphreys 1996] Watts Humphreys, *Introduction to the personal software process* ADDISON-WESLEY; December 1996. ISBN-13: 978-0201548099

[Humphreys 1989] Watts Humphreys, *Managing the Software Process*, ADDISON-WESLEY PROFESSIONAL; January 11, 1989. ISBN-13: 978-0201180954

[Kelly 1970] T.J. Kelly, *The dynamics of R & D project management*, Master's Thesis, Sloan School of Management, Massachusetts Institute of Technology, 1970.

[Lakos 1996] John Lakos, *Large-Scale C++ Software Design*, ADDISON-WESLEY PROFESSIONAL; July 20, 1996. ISBN-13: 978-0201633627

[Lehman 1998] Meir M. Lehman, *Softwares Future: Managing Evolution*, IEEE SOFTWARE; Vol. 15, no. 1, pp. 40-44 January 1998.

[Lyneis, Cooper and Elsa 2001] James M. Lyneis, Kenneth G. Cooper and Sharon A. Elsa *Strategic management of complex projects:a case study using system dynamics*, SYSTEM DYNAMICS REVIEW; Vol. 17, no. 3, pp. 237-260 Fall 2001.

[Reichelt & Lyneis 1999] Kimberly S. Reichelt, James M. Lyneis *The Dynamics of Project Performance: Benchmarking the Drivers of Cost and Schedule Overrun* EUROPEAN MANAGEMENT JOURNAL; Vol. 17, No. 2, pp. 135-150 April 1999.

[Lyneis 2012] James M. Lyneis, *Lecture notes for ESD.36 - System Project Management* MASSACHUSETTS INSTITUTE OF TECHNOLOGY; unpublished, Fall 2012.

[McConnell 1996] Steve McConnell, *Rapid Development: Taming Wild Software Schedules*, MICROSOFT PRESS; 1996. ISBN: 978-1556159008.

[McConnell 1997] Steve McConnell, *Software Project Survival Guide*, MICROSOFT PRESS; October 22, 1997. ISBN-13: 978-1572316218

[McConnell 2004] Steve McConnell, *Code Complete: A Practical Handbook of Software Construction* MICROSOFT PRESS; July 7, 2004. ISBN-13: 978-0735619678

[MacCormack 2006] A. D. MacCormack, J. Rusnak, and C. Y. Baldwin, *Exploring the structure of complex software designs: An empirical study of open source and proprietary code*, MANAGEMENT SCIENCE, pp. 1015-1030; 2006.

[MacCormack, Baldwin, & Rusnak 20121] Alan MacCormack, Carliss Y. Baldwin, and John Rusnak, *Exploring the Duality Between Product and Organizational Architectures: A Test of the 'Mirroring' Hypothesis*, RESEARCH POLICY; vol 41, no. 8, pp. 13091324. October, 2012.

[Meyer 2007] Marc H. Meyer, *The Fast Path to Corporate Growth: Leveraging Knowledge and Technologies to New Market Applications*, OXFORD UNIVERSITY PRESS; 2007. ISBN-13: 978-0195180862

[Münch, Armbrust, Kowalczyk & Soto 2012] Jürgen Münch, Ove Armbrust, Martin Kowalczyk, Martín Soto, *Software Process Definition and Management*, SPRINGER; 2012. ISBN: 978-3-642-24290-8

[Nay 1965] J.N. Nay, *Choice and allocation in multiple markets: a research and development systems analysis*, Master's Thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, 1965.

[Novak & Levine 2010] William Novak & Linda Levine, *Success in Acquisition: Using Archetypes to Beat the Odds (CMU/SEI-2010-TR-016).* SOFTWARE ENGINEERING INSTITUTE, Carnegie Mellon University; 2010.

[Parnas 2001] David Parnas, *Software Fundamentals: Collected Papers By David L. Parnas* ADDISON-WESLEY PROFESSIONAL; April 2001. ISBN-13: 978-0201703696

[Paulk, Curtis, Chrissis & Weber 1993] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, Charlie Weber, *Capability Maturity Model for Software*, TECHNICAL REPORT CMU/SEI-93-TR-024; February 1993

[Pepin 1999] Ronald Pepin, *Application of Critical Chain to Staged Software Development*, MIT SDM Thesis; January 1999.

[Pohl, Böckle, & van der Linden 2010] Pohl, K., Böckle, G., and van der Linden, F. J. *Software Product Line Engineering: Foundations, Principles and Techniques*, SPRINGER; November 9, 2010. ISBN-13: 978-3642063640

[Pressman 2000] Roger Pressman, *Software Engineering: A Practitioner's Approach*, MCGRAW-HILL; December 2000. ISBN-13: 978-0073375977

[Rahmandad & Weiss 2009] Hazhir Rahmandad, David M. Weiss, *Dynamics of concurrent software development*, SYSTEM DYNAMICS REVIEW; Vol. 25, No. 3, (JulySeptember 2009): pp 224249.

[Rahmandad & Hu 2010] H. Rahmandad and K. Hu, *Modeling rework cycle: comparing alternative formulations*, SYSTEM DYNAMICS REVIEW; Vol. 26, No. 4, pp 291-315.

[Rechtin 1990] Eberhardt Rechtin, *Systems Architecting: Creating & Building Complex Systems*, PRENTICE HALL; December 1, 1990. ISBN-13: 978-0138803452

[Repenning 2001] Nelson Repenning, Paulo Gonçalves, Laura J. Black, *Past the Tipping Point: the persistence of Firefighting in Product Development*, MIT SLOAN SCHOOL OF MANAGEMENT; 2001.

[Royce 1987] Winston W. Royce, *Managing the development of large software systems: concepts and techniques*, IEEE COMPUTER SOCIETY PRESS; ICSE '87 Proceedings of the 9th international conference on Software Engineering, p 328-338, 1987. ISBN:0-89791-216-0

[Russell 2007] Gregory B. Russell, *A Systems Analysis of Complex Software Product Development Dynamics and Methods* MIT SDM Thesis; September 2007.

[SEI 2013] Software Engineering Institute, http://www.sei.cmu.edu/productlines/ Fetched on April 18th, 2013.

[Shaw, Howatt, Maness & Miller 1989] Shaw, W.H., Jr. ; Howatt, J.W. ; Maness, R.S. ; Miller, D.M. *A software science model of compile time*, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING; Vol. 15, no. 5, pp. 543-549, May 1989.

[Simon 1962] Herbert A. Simon, *The Architecture of Complexity* PROCEEDINGS OF THE AMERICAN PHILOSOPHICAL SOCIETY; Vol. 106, No. 6., pp. 467-482. Dec. 12, 1962.

[Stark 2000] John Stark, *Global Product: Strategy, Product Lifecycle Management and the Billion Customer Question*, SPRINGER; August 24, 2007. ISBN-13: 978-1846289149

[Sterman 2000] John D. Sterman, *Business Dynamics: Systems Thinking and Modeling for a Complex World*, MCGRAW-HILL/IRWIN; February 23, 2000. ISBN-13: 978-0072389159

[Sturtevant 2013] Daniel J. Sturtevant, *System Design and the Cost of Architectural Complexity.* MIT PhD Thesis, February 2013.

[Sullivan 2001] Ed Sullivan, *Under Pressure and On Time*, MICROSOFT PRESS; May 4, 2001. ISBN-13: 978-0735611849

[Trammell, Madnick & Moulton 2012] Travis Trammell, Stuart Madnick, Allen Moulton *Effect of Funding Fluctuations on Government Funded Software Development*, MIT Sloan Working Paper CISL 2012-08, June 2012.

[Tulach 2012] Jaroslav Tulach, *Practical API Design: Confessions of a Java Framework Architect* APRESS; June 6, 2012. ISBN-13: 978-1430243175

[Weinberg 1991] Gerald M. Weinberg, *Quality Software Management: Systems Thinking*, DORSET HOUSE; September 1991. ISBN-13: 978-0932633224

[Wheelwright & Clark 1994] Steven C. Wheelwright, Kim B. Clark, *Leading Product Development: The Senior Manager's Guide to Creating and Shaping the Enterprise*, FREE PRESS; October 1, 1994. ISBN-13: 978-0029344651

[Xitong & Madnick 2012] Xitong Li & Stuart Madnick, *Understanding the Organizational Traps in Implementing Net-Centric Systems*

[Yourdon 1989] Edward Yourdon, *Structured Walkthroughs*, YOURDON; January 1989. ISBN-13: 978-0138552893