Agile Project Dynamics:
A Strategic Project Management Approach to the Study of Large-Scale Software Development Using System Dynamics

Firas Glaiel

Working Paper CISL# 2012-05

June 2012

Composite Information Systems Laboratory (CISL)
Sloan School of Management, Room E62-422
Massachusetts Institute of Technology
Cambridge, MA 02142

Agile Project Dynamics: A Strategic Project Management Approach to the Study of Large-Scale Software Development Using System Dynamics

By

Firas Glaiel
B.S., Computer Systems Engineering
Boston University, 1999

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 2012

Signature of Author: ____
System Design and Management
May 7, 2012

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

This page left intentionally blank

Agile Project Dynamics:
A Strategic Project Management Approach to the Study of Large-Scale Software Development Using System Dynamics

By
Firas Glaiel

Submitted to the System Design and Management Program on May 18th, 2012 in Partial Fulfillment of the Requirements for the Degree of Master of Science in Engineering and Management

Abstract

The primary objective of this research is to understand the dynamics of software development projects employing the Agile approach. A study of several Agile development methodologies leads us to the identification of the "Agile Genome": seven characteristics that Agile projects share. We gain insight into the dynamics behind Agile development by constructing a System Dynamics model for Agile software projects, called the Agile Project Dynamics (APD) model, which captures each of the seven genes as a major component of the model.

Large-scale software engineering organizations have traditionally used plan-driven, heavyweight, waterfall-style approaches for the planning, execution, and monitoring of software development efforts. This approach often results in relatively long development schedules that are susceptible to failure, especially in a rapidly changing environment: Schedule pressure, defects and requirements changes, can drive endless redesign, delay the project, and incur extra cost. Many in the commercial software world have dealt with these pressures by adopting Agile Software Development, an approach designed to be flexible and responsive to high-change environments.

Software development teams that are said to employ "Agile development" in effect practice a variety of "agile methods". These practices are advertised to reduce coordination costs, to focus teams, and to produce stable product iterations that can be released incrementally. Agile software development has become a de-facto approach to the engineering of software systems in the commercial world, and is now entering the aerospace and defense sectors.

The APD model developed in this research aids in the understanding of the impact that alternative combinations of Agile practices, combined with different management policies, have on project performance, compared to a waterfall approach. This research culminates in a formulation of insights and recommendations for how to integrate Agile practices into a large-scale software engineering organization.

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

This page left intentionally blank

Acknowledgements

I'd like to thank Stuart Madnick, my advisor, for his guidance and direction in completing this research. As a member of Dr. Madnick's Information Technologies Group in the Sloan School of Management, I have had the incredible opportunity to collaborate with and learn from some of the brightest minds on the planet. A special thank you also goes to Allen Moulton, research scientist at the MIT Engineering Systems Division, whose encyclopedic domain knowledge and valuable insights contributed greatly to the quality of this work. I would also like to express my appreciation to our project sponsors, especially Mr. Richard Herrick at the Defense Intelligence Agency and Mr. Douglas Marquis at MIT Lincoln Labs, for their valuable insights into the thesis problem space from both the client and the contractor side of large-scale software engineering projects.

Thank you to Patrick Hale and the wonderful staff at the MIT System Design and Management (SDM) program. SDM's interdisciplinary curriculum, taught by faculty experts from all departments in MIT's School of Engineering and MIT Sloan, has transformed my mindset. It has armed me with a Systems Thinking perspective that integrates management, technology, and social sciences; and with leadership competencies to address rapidly accelerating complexity and change across organizational and cultural boundaries. SDM is truly a once-in-a-lifetime transformative experience.

This effort would not have been possible without the support of my management at Raytheon Network Centric Systems. I'd like to thank Bronwen Bernier, Tony Casieri, and Edmond Wong for their support and patience during the two-plus years that I have been supporting the business part time while completing my graduate studies.

Last but not least, I'd like to thank my family, especially my wife Kimberly who has had to cope with a part-time husband during a period where I have had to juggle work, school, and being a new parent. Her love and support are what ultimately drove me to go back to school and to get through the hard times. Thank you Kim for making me take the GRE exam (unprepared) three years ago, and submitting my application to SDM– without you I would not be where I am today.

This page left intentionally blank

Table of Contents

1. Introduction

1.1 Research Motivation

The software industry is experiencing a high-velocity market environment. This is especially true in the U.S. government software sector, be it in government's acquisitions of Information Technology (IT) systems, aerospace and defense systems, transportation systems, or other. Government customers are demanding higher productivity and faster cycle times for the delivery of software services, capabilities, and products. This, in turn, is causing software contractors to look beyond the traditional "Waterfall" approach to software engineering, and consider the so-called "agile" development methodologies that, in the commercial software world, have been heralded as being *the* way of doing things.

Traditional software engineering approaches are said to have helped government contractors control schedules and costs for large-scale software projects. By "control" we mean "adhere to plan." If a multi-year software project meets all of its schedule milestones and cost targets, then it is said to be a success. If all of a software firm's projects enjoyed such success, then the organization can be said to be "predictable" and to have "repeatable" processes: it can be trusted to deliver software as planned (cost and schedule-wise.) This is a characteristic that is desirable from a customer's point of view.

But the promise of Agile is that it will help software development organizations *cut* costs and *shorten* development times. Process-heavy/Waterfall software engineering is blamed by some for inflating the costs and duration of software projects. It can be argued that these process-laden development practices have redirected the focus of software teams towards statistical process control, driven by the desire to achieve project predictability and higher Capability Maturity Model (CMM) ratings as described in section 2.3, and to fit into established models of project management. Some may even argue that this takes away from a firm's focus on faster time-to-market, greater customer satisfaction, better design and architecture quality, etc.

Today's government customer calls for agility and responsiveness; the ability for rapid application deployment in the face of unpredictable and changing requirements. There is no better example of a high-volatility requirements environment than the war fighter's situation: the threat is constantly evolving, and the fighter's needs and requirements for intelligence capabilities are changing at a rate measured by weeks and months, not years. Under such circumstances a level-5 CMM rated organization serves them little if it cannot quickly deliver software-based capabilities to fulfill a real need, in time, and evolve the software as the need changes.

As the government agencies have begun to demand faster development times and lower costs, the contractor community that serves them has turned towards legitimately looking at Agile development, with an eye towards delivering capabilities at the pace and cost needed to stay competitive in the government software market. In the commercial world, where being first-to-market and having the ability to understand and cater to your customer's evolving needs can mean the difference between life and death for a small software startup, organizations have embraced the family of agile methodologies as the way to better flexibility, greater responsiveness, and improved time-to-market.

Fred Brooks, in his seminal paper on software engineering, *No Silver Bullet — Essence and Accident in Software Engineering*, argues that there is no, and probably never will be any "silver bullet" to solve the problem of software project failures, and that building software will always be a very difficult endeavor (Brooks 1987). One of the characteristics of modern software systems that he points to as being at the root of the problem is complexity. Complexity results in communication difficulties (among software teams) and associated managerial difficulties in the planning and execution of software projects. Agile development methods are advertised to mitigate this problem by employing complexity-reducing practices, such as the break-up of the software into manageable "features" developed in short "sprints", and improved flow of communication through frequent face-to-face meetings.

Although Agile is the topic du-jour in the software community, many new technologies and methodologies in the past have also been acclaimed as the "the silver bullet" but have ultimately failed to meet expectations. In his IEEE article *Software Lemmingineering*, Alan Davis observes that time and time again, software organizations tend to blindly follow certain practices just because the masses are adopting them, and points out that we should be wary of such "lemmingineering." He lists several fads that have stricken the software development community; he writes: "At this year's ICSE (International Conference on Software Engineering) I got the impression that everyone has at least one foot in the process maturity stampede." (Davis 1993) Interestingly, this was written in 1993, when the CMM was gaining momentum as the premiere approach to software engineering in industry. We have now come full circle, and Agile is the hot topic in software engineering circles. Is this another fad or can agile methods truly be adopted and implemented within the rigid structure of large-scale software engineering firms?

1.2 Personal Experience with Agile

During my twelve-plus years of experience as a software engineer in one of the largest aerospace and defense contractors (Raytheon,) I have worked on several software-intensive programs mainly in the areas of Ballistic Missile Defense Systems (BMDS) and Air Traffic Management (ATM.) These were all large mission-critical, software-intensive systems, and often involved teams of over one hundred engineers. During this time, the computer systems and software engineering industry underwent many technological shifts in various aspects of hardware (processing speed and power, memory, peripherals,) software stacks (operating systems, programming languages, and tools) and with respect to software development processes and methodologies. The following is a description of personal experiences and insights, representing my own personal views. In no way do they represent any official views or positions of or by Raytheon.

1.2.1 Early Experience with Agile Methodologies (1999-2001)

A large part of my time as a software engineer was spent developing software for the Standard Terminal Automation Replacement System (STARS.) This is a joint Federal Aviation Administration (FAA) and Department of Defense (DoD) program aimed at replacing aging ATM systems across the country with a state-of-the-art solution for Air Traffic Control (ATC.) The STARS system receives and processes target reports, weather, flight plan information, and other data from a variety of radars, digital sensors, and external systems. It tracks aircraft and other "surveillance targets" and displays their position information to air traffic controllers on large graphical displays, provides decision support tools, data analysis tools, as well as visual aids and safety functions such as the detection of unsafe proximities between aircraft and warnings if aircraft are detected at a dangerously low altitude. The STARS system (hardware and software) has been evolving for almost two decades, controlling air traffic at a majority of the nation's airports.

In 1999, the FAA was funding new system enhancements which required STARS to communicate with a variety of other systems supporting the FAA's operations in the National Air Space (NAS). An "Application Interface Gateway" (AIG) subsystem was called for to provide these external communication functions. At the time, there were several difficulties on the road to putting together a new development team for the AIG: Program management had agreed to a very aggressive schedule and the program was challenged to produce the full scope of software on time and within budget. Engineering, on the other hand, could not see a way to make this effort fit the schedule, based on traditional multi-phased waterfall approach, and the lengthy schedule outlook. It simply couldn't be done, given the way we did business traditionally: We did not have the ability to quickly implement software changes and enhancements.

Analysis showed that slow reaction time was mainly due to the sheer weight of the software development process: System requirements were broken down into several tiers of subsystem requirements, against which a software design was produced, with elaborate code-to-requirements tracing. This was followed by Detailed Design which consisted of writing software modules in pseudo-code like language called PDL (Program Description Language), and finally coding in the C programming language. Unit testing was a long and tedious manual process that included the writing of detailed test procedures for every software module.

There were several process-related meetings and peer reviews at each afore-mentioned stage. Therefore there was legitimate concern as to our ability to develop the AIG within the time and cost constraints of the program. Even worse, all of the senior engineers were busy with other critical system enhancements and could not be dedicated to this particular thread of development.

At the time we had new and forward-thinking management, who were not afraid to adopt new methods and sponsor innovative thinking. Conditions were ripe for exploring a novel approach to software development. With software management's support and sponsorship (including funding) we set forth to find an approach that would be better suited for this endeavor. Another goal in mind was to introduce new programming languages and design approaches. Specifically, the program was interested in introducing Object-Oriented (OO) design and coding, and languages such as C++. This was the time when Object Oriented approaches to software were "hot" and gaining momentum in industry, as opposed to traditional "procedural" (also known as "imperative") and "functional" programming styles. There was difficulty in hiring new engineers proficient in procedural languages such as C at that time, when most graduates were taught OO design approaches and languages (e.g. Java, C++) and web-based technologies. This was after all, during the time of the internet boom.

Some of the main challenges were: how to capture software requirements for an OO software product? How do you document design for OO software? Is C++ a viable programming language? What would the development process be? First, a small team of engineers went through training provided by the Rational Corporation (now part of IBM) where we were taught the Rational Unified Process (RUP) – This training consisted mostly of learning the Unified Modeling Language (UML) and of learning to use Rational Rose, a UML modeling tool.

Additionally, I led research into several new (at the time) concepts for software development which today fall under the general banner of "Agile Methods" (note that the term "agile" was not yet popular or used in industry in the late 1990s)– I focused specifically on "eXtreme Programming." Keep in mind that this was 1999, and OO had not yet been used at all, in this part of the company, in any delivered system or product. It was important to define a software development process for OO.

The existing stove-piped process was very specific and tied to the C language and left no room for OO practices. After two months of research and training, we took what we found to be the best practices from RUP and eXtreme Programming, and other emerging concepts (such as automated unit-testing) to draft what we eventually named the "PrOOD" (Process for Object Oriented Development). Some of the highlights in this process are: pair programming, team collocation, iterative development cycles, model-based automatic code generation, and automated unit testing. These place a focus on and help in the early identification of software defects, and also speed up the development cycle.

Armed with the PrOOD as our official development process, a small team of 6 engineers produced in months what may have taken over a year to do: The requirements and design of the AIG were completely defined in a UML model using Rational Rose (requirements were modeled as Use Cases). The code was written in C++, and automated Unit Test framework was developed.

Over time, the AIG has evolved to be one of the key STARS components, especially in the net-centric ATM world. Adding software interfaces to the system has become a relatively simple process. Almost a decade later, for example, the AIG was used to connect STARS to ADS-B (Automatic Dependent Surveillance-Broadcast) data sources, allowing GPS-based tracking (as opposed to radar-only tracking) of aircraft, enabling satellite-based air traffic management, which was a huge leap forward in the FAA's modernization efforts. This functionality went live in the Philadelphia airport in April of 2010, and is expected to be deployed nationwide by 2013, allowing air traffic controllers to efficiently and safely track and separate aircraft.[1]

As a software component, the AIG also has the lowest defect density of all CSCIs (Computer Software Configuration Items, i.e. components) in the product line. Defect density is measured by number of software defects found per one thousand lines of code, post-release. The PrOOD itself, which was an innovative process for Object Oriented design and development, found its way eventually to become part of the organization's standards and practices (referred to as the 'engineering blue book') on how to build object-oriented software. The blue book was soon superseded by a "Common Process Architecture," (CPA) during the CMM adoption frenzy that characterized software engineering firms of the early 2000s (more on CMM later). Meanwhile, the PrOOD was forgotten and our successes and experiences with agile development were lost in the sands of time.

1.2.2 Recent Experience with Scrum (2010-2011)

In 2010, a new opportunity to employ agile development presented itself: the FAA was seeking to upgrade their terminal ATC systems' data recording capabilities. The deployed systems were recording data on outdated storage media. Specifically, data was recorded onto magnetic tapes using DAT (Digital Archive Tape) drives, and the FAA wanted to upgrade the system and software to perform data recording onto RAID (Redundant Array of Independent Disks) storage devices, for many appropriate reasons such as reliability, redundancy, capacity, and performance. The effort was dubbed "CDR-R" for Continuous Data Recorder Replacement. The software changes required to make this transition were complex and non-trivial, requiring modifications to portions of the baseline code, some of which dated back to the 80s. Data recording is a critical component of the system – the FAA takes it very seriously, as they should, since this data is used not just for incident investigation purposes, but also to support data analysis of FAA operations.

The problem was that the scope of the changes that we estimated would be needed to implement the solution were such that they did not fit into the customer's yearly budget and schedule… This seemed to me to be an opportunity to try a new delivery approach: First, inspired by Feature Driven Design (FDD,) I proposed that we divide the scope of changes into a set of 'features' such as 'RAID recording', 'RAID synchronization', 'data transfer', etc. Then, inspired by the spiral development model (described in section 3.3.2), I proposed an incremental delivery of the CDR-R changes in three phases: the first phase would provide basic RAID-based recording capabilities, the second would add additional monitoring and control capabilities for the RAIDs and provide data transfer tools to allow migration of data across various media and system installations, and finally the third phase would add some data maintenance functions. This new (in our program's context) delivery model made the CDR-R transition more palatable for our customer and made for a better fit

[1] http://www.aviationtoday.com/categories/commercial/67917.html

with their funding cycles, leading them to authorize the CDR-R development to begin.

For the first phase's set of features, we employed our traditional waterfall approach: software requirements were developed to meet the system level requirements, and then a design to fulfill the software requirements, then code was developed to implement the design, then testing and integration. This phase completed with a software delivery containing the full scope of planned phase-1 modifications, however the performance of the development effort suffered from cost overruns and internal schedule slippage. A lot of rework was generated towards the end of the development phase, as we began to test the software and encountered complications when running in the target hardware environment. Several other problems required revisiting the requirements, the design, and underlying code implementation. In my opinion, at the root of these issues were two main root causes: Unclear/uncertain requirements, and a delayed ability to test the software.

These two issues have a compounding effect on each other: due to the waterfall approach of developing components in isolation of each other, the system functionality that we were trying to achieve only came to being at the tail-end of the development process, when all of the CSCIs (software components) were developed, tested in isolation, then integrated to produce the required functionality. Unfortunately, we typically only reach this point after 3/4ths or more of the schedule and budgeted effort have been consumed. This may not be problematic if requirements are well-defined, correct, complete, and unambiguous – but here we found this not to be the case. Identifying requirements issues that late in the development process meant several re-water-falling rework iterations of requirements/design/code at the very end of the development cycle when little time was left. Two key project performance measures, Schedule Performance Index (SPI) and Cost Performance Index (CPI), for this phase-1 of development were 0.82 SPI and 0.76 CPI, meaning that the software was behind schedule by 18%, and 24% over cost.

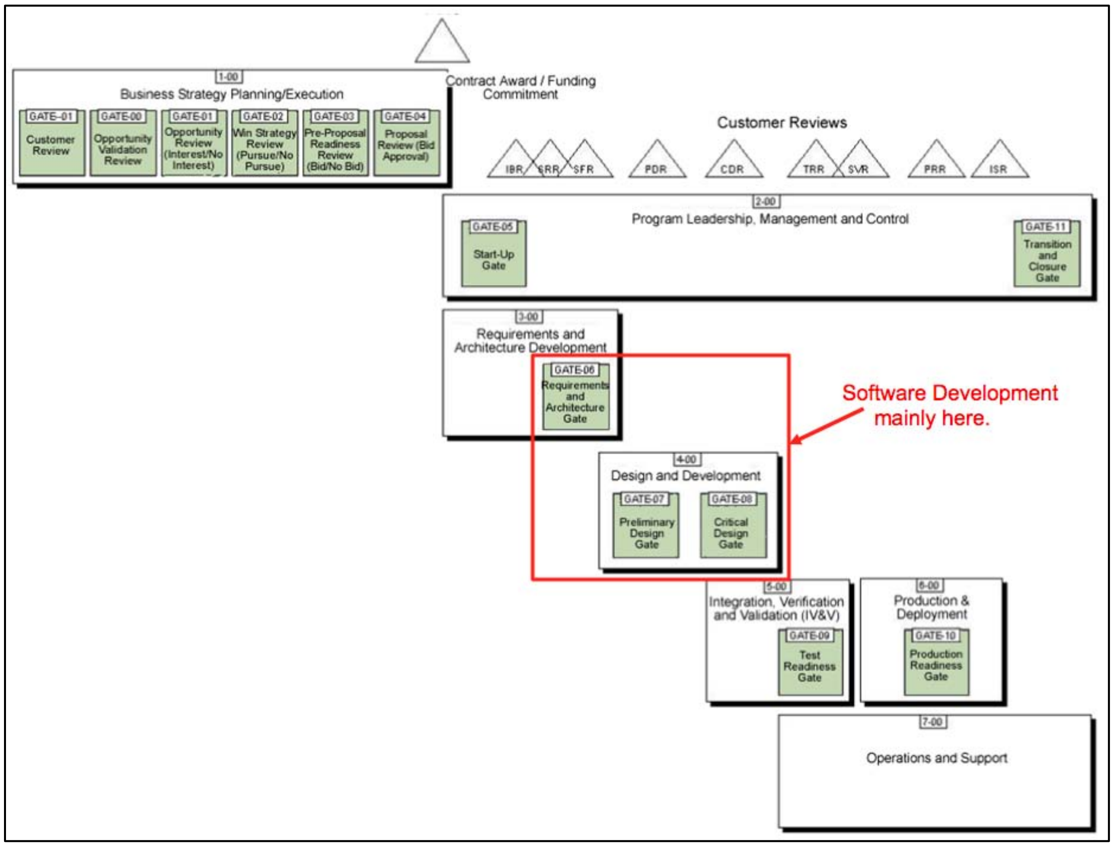
For the second phase, I proposed that we employ an Agile approach to development, based on the Scrum methodology (described later in this research.) Management was receptive to this proposition, and eager to support this effort, especially as the desire to "go agile" was being communicated top-down through the organization, exemplified by a corporate-backed initiative named "SWIFT" (Software Innovation for Tomorrow) that had begun piloting Agile practices on several other large programs in the company. Note that ours was not one of the SWIFT pilot programs, but an organic home-grown desire to employ Agile methods, with prior successes in AIG development and the PrOOD in mind.

A small team of developers was assembled, trained in Scrum methodology, and produced the phase-2 software increment in three sprints of three weeks each. This is relatively fast in an organization where conducting a code inspection alone can eat up a week in development time. Phase-2 thus enjoyed an SPI of 1.07 (7% ahead of schedule) yet the CPI of 0.73 remained less than ideal.

While the Scrum approach helped us beat schedule, our cost performance did not improve. Looking back, there are several factors that could explain this: The team was new to Scrum, required training sessions, and only had the chance to execute three sprints. We did not have a product owner or customer representatives involved so the requirements were still unclear and conflicting. A new web-based code review tool was deployed during our second sprint, but in retrospect was not optimally used and ended up causing wasted effort. We also had to deal with a slew of defects and issues from the phase-1 delivery; in other words we were building on top of a baseline that still had undiscovered defects. Finally, there was the overhead of trying to fit the agile software development model within an overarching heavyweight Integrated Product Development Process (IPDP) covering all of the phases of program execution.

1.2.3 Personal Reflections

It is important to note that the activity of "software development" fits within an integrated approach to product development that includes program processes such as business planning and requirements analysis, as shown in Figure 1.



Contract Award / Funding Commitment
1-00
Business Strategy Planning/Execution
GATE-01 Customer Review
GATE-00 Opportunity Validation Review
GATE-01 Opportunity Review (Interest/No Interest)
GATE-02 Win Strategy Review (Pursue/No Pursue)
GATE-03 Pre-Proposal Readiness Review (Bid/No Bid)
GATE-04 Proposal Review (Bid Approval)
Customer Reviews
IBR
SRR
SFR
PDR
CDR
TRR
SVR
PRR
ISR
2-00
Program Leadership, Management and Control
GATE-05 Start-Up Gate
GATE-11 Transition and Closure Gate
3-00
Requirements and Architecture Development
GATE-06 Requirements and Architecture Gate
Software Development mainly here.
4-00
Design and Development
GATE-07 Preliminary Design Gate
GATE-08 Critical Design Gate
5-00
Integration, Verification and Validation (IV&V)
GATE-09 Test Readiness Gate
6-00
Production & Deployment
GATE-10 Production Readiness Gate
7-00
Operations and Support

Figure 1 - Example Integrated Product Development Process

Part of the problem with adopting Scrum within an environment such as shown in Figure 1 was that, as a Software Development Manager, only software development activities were within my purview, and thus only the software development team could adopt Scrum. Other parts of the engineering organization, such as System Engineering (SEs, producers of software requirements) and Software Integration and Test (SIs, integrators and testers of the system) were still operating in a traditional approach, where SEs hand off completed requirements to development, who in turn hand off completed code to SIs.

Since Scrum was adopted just within the confines of software development, and not as a complete IPDP overhaul, this meant that the development team had to maintain the same "interfaces" to the other activities in the program, and had to produce the same work artifacts (e.g. design documentation, review packages), employ the same level of process rigor– all of which was monitored by the Software Quality Assurance (SQA) oversight group. This included holding all of the required inspections and feeding back progress to management, which expected clear Earned Value (EV) based reports of progress.

Yet with all of this, Scrum seemed to energize the development team, produce working code much faster, and speed up the experience gain of new engineers. Clearly there was some benefit to Agile. As a result I decided to pursue this research on Agile development for my graduate thesis in the M.I.T. Systems Design & Management (SDM) program. The research would take a "deep dive" into Agile, and take a close look at agile practices with the goal of understanding how to integrate them in a CMMI level 5 software engineering environment. During my time at SDM I also discovered the worlds of Systems Thinking and Systems Dynamics (SD), and found the SD approach to be the best tool for understanding the emergent behavior of a 'complex socio-technical system', in this case the software development enterprise.

1.3 Contributions

The scientific contribution of this work is in developing the “Agile Genome”, a framework for understanding the nature of Agile software development, from a project management perspective. This understanding of the Agile Genome leads us to the development of the Agile Project Dynamics (APD) model which can be used as a tool for experimentation and learning about the behavior of the software project-system under various project conditions and with a variety of management policies and decisions. Additionally, this work demonstrates the value of the System Dynamics methodology for the modeling and simulation of complex socio-technical systems.

1.4 Research Approach and Thesis Structure

This research aims to understand the dynamics that drive the performance of Agile software projects, and the strengths and weaknesses of “Agile” as compared to the classic software engineering approaches that are based on the Waterfall model. Software development projects are in and of themselves complex socio-technical systems whose behavior is driven by the interactions of people, processes, tools, and policies. When we add layers of planning, budgeting, staffing, and management of such a program, it becomes an even more complex and dynamic system with many competing feedback effects. This section details the research approach we have employed to study the software project-system, and presents the structure of this research chapter-by-chapter.

Chapter 2 presents a brief review of relevant topics in software project management, including a historical review of waterfall software development and the rise of Agile methodologies. The chapter concludes by looking at why large-scale software engineering organizations, traditional laggards in the adoption of now processes and methodologies, are now embracing Agile development.

Chapter 3 presents an overview of Agile software development and leads to an understanding of the nature of agility in software projects. Several popular methodologies are inspected in order to distill the essence of what makes a software project “agile” - We call the results of this analysis the “Genome of Agile Development.” This is based on extensive literature review, interviews, and professional experience with software development teams.

To frame and understand the behavior of “the software development project as a system”, this research will employ a holistic view of the software project-system and will capture its structure and feedback effects in the form of a System Dynamics model.

Chapter 4 presents System Dynamics (SD), a powerful methodology for framing, understanding, and discussing complex strategic project management issues and problems. Software project management is well suited for a "systems approach" because those involved in software development must not only understand the minutia of software development technologies, processes and tools, but also the complexities of project-team dynamics, as well as the effects of management policy (especially the far-reaching effects of short-term management actions and decisions.) System Dynamics provides a method to model the cause-and-effect relationships among various policy variables.

Chapter 5 follows by constructing a system dynamics model to simulate the effects of agile practices on software project performance, using the insights gained from the previous chapter. The modeling is inspired by the work of Abdel-Hamid and Madnick, in *Software Project Dynamics*, which includes a validated "Software Project Dynamics" (SPD) model of the classic Waterfall process. We create an "Agile Project Dynamics" (APD) to study the strengths and weaknesses of the Agile approach as compared to the classic Waterfall model. We use this APD model to understand how and why the project-system behaves as it does. It can be used to understand the impact of management policies and adoption of one, many, or all of the "Agile genes".

In **Chapter 6**, the APD model is used to perform "what if" scenario investigation, by performing several controlled simulation experiments and observing project performance under several policy variable combinations. We start with a base-case single pass waterfall project and compare its behavior to that of an Agile project with several combinations of Agile genes.

Finally, **Chapter 7** concludes with a set of insights and observations gained from out research and experimentation with the model, which can lead to the formulation of good rules and policies for the management of software development projects.

2. A Brief Review of Relevant Software Engineering Topics

Large-scale software engineering organizations, particularly government contractors, have traditionally used plan-driven, heavyweight, waterfall-style approaches to the planning, execution, and monitoring of large software development efforts. This approach is rooted in ideas stemming from statistical product quality work such as that of W. Edward Deming and Joseph Juran.

The general idea is that organizations can improve performance by measuring the quality of the product development process, and using that information to control and improve process performance. The Capability Maturity Model (CMM) and other statistical/quality–inspired approaches such as Six Sigma and ISO 9000 follow this idea. As a result, the collection and analysis of process data becomes a key part of the product development process (Raynus 1998).

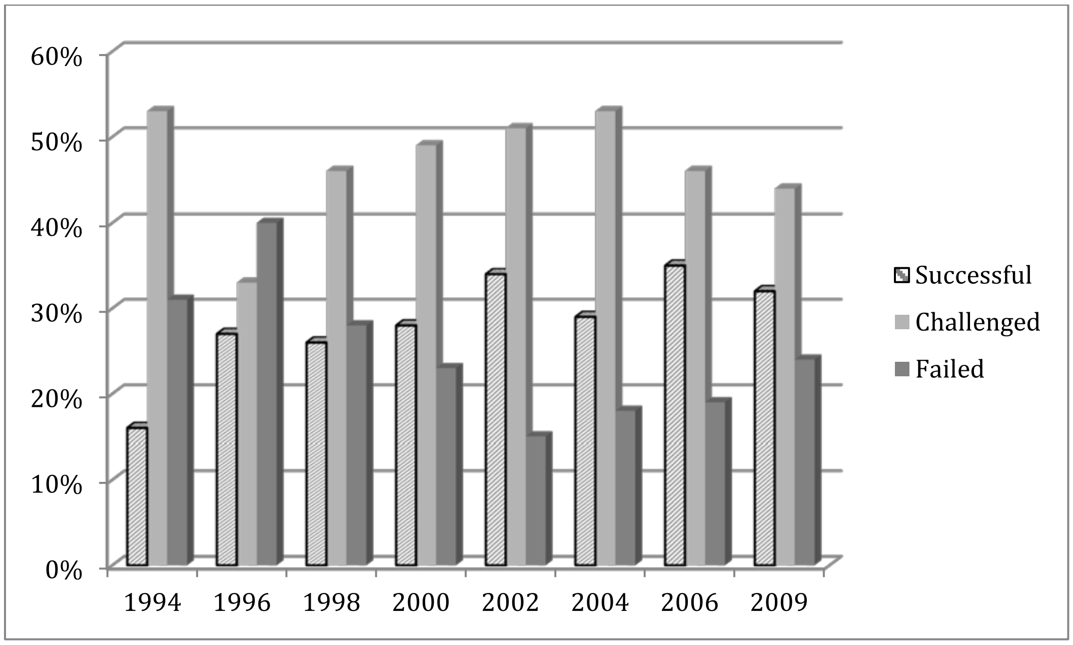


Figure 2 - CHAOS Report Summary 1994-2009

Nevertheless, these types of “big” software projects have been susceptible to failure in terms of cost, schedule, and/or quality. The Standish group every few years produces the “CHAOS Report,” one of the software industry’s most widely-cited reports showcasing software project success and failure rates. It measures success by looking at the “Iron Triangle” of project performance: schedule, cost, and scope (whether or not the required features and functions were delivered.) Roughly, only about a third of software projects are considered to have been successful over the last two decades (see Figure 2) In April 2012, the worldwide cost of IT failures was conservatively calculated to be around $3 trillion (Krigsman 2012).

It is with this in mind that the software industry often refers to "the software crisis." A whole business ecosystem has evolved around the software industry's need to address the software crisis, including:

- Project management approaches to controlling schedule, cost, and to building more effective teams.
- Engineering approaches to architecture and design to produce higher quality (low defect, more flexible, more resilient, scalable, modifiable, etc.) software.
- Processes and methodologies for increasing productivity, predictability, and efficiency of software development teams.
- Tools and environments to detect and prevent defects, improve design quality, automate portions of the development workflow, and facilitate team knowledge.

Let us briefly explore some of these in the context of government software systems development.

2.1 Software Project Management

2.1.1 The Iron Triangle

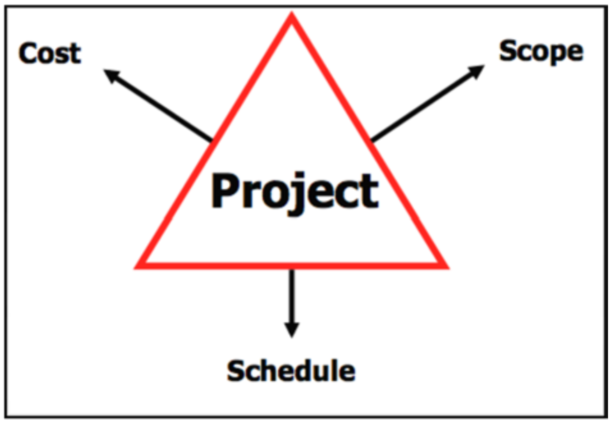


Figure 3 - The Iron Triangle of Project Management

Like any human undertaking, projects need to be performed and delivered under certain constraints. Traditionally, these constraints have been listed as "scope," "time," and "cost" (Chatfield & Johnson 2007). Note that "scope" in this context refers to the set of required features in the software, as well as the level of quality of these features.

These three constraints are referred to as the "Iron Triangle" (Figure 3) of project management – This is a useful paradigm for tracking project performance: In an ideal world, a software project succeeds when it delivers the full scope of functionality, on schedule as planned, and within budget. However, in the real world projects suffer from delays, cost overruns, and scope churn. Expressions such as "scope churn" and "scope creep" refer to changes in the project's initial scope (often captured contractually in the form of requirements). A project with many Change Requests (CRs) is said to experience high scope churn. In order to deliver, project managers must adjust project goals by choosing which sides of the constraints to relax. When a project encounters performance problems, the manager's choices are to pull one of the three levers of the iron triangle:

- Increase effort (and thus cost) by authorizing overtime or hiring more staff.
- Relax the schedule by delaying delivery or milestones.
- Cut scope by deferring a subset of features to future software releases, or reduce the quality of the features delivered.

As will be shown later in this research, taking any one of these actions can result in negative and unforeseen side-effects, making things even worse. In section 4.3, we will take a close look at "Brooks' Law" which states that "adding manpower to a late software project makes it later".

2.1.2 Earned Value Management

Earned Value Management (EVM) is a management methodology for monitoring and controlling schedule, cost, and scope. It allows management to measure and track project execution and progress. In 1997 the U.S. Office of Management and Budget (OMB) released the *Capital Programming Guide*, which since then requires the use of EVM for all contractor performance-based management systems. This guide is an appendix to the OMB's Circular A-11, which provides guidance on preparing Fiscal Year (FY) budgets and contains instructions on budget execution. This means that all budget approvals depend on performance as measured by EVM, behooving government contractors such as Raytheon, Boeing, Lockheed Martin, General Dynamics, Northrop Grumman, and others to adopt EVM as part of their standard management practices. EVM evolved from the DoD Cost/Schedule Control Systems Criteria (C/SCSC) which was an early attempt in 1967 to standardize contractor requirements for reporting cost and schedule.

The problem with comparing actual expenditures to baseline plan, which is what other project management techniques do, is that it ignores the amount of work "actually completed." EVM addresses this by distinguishing between cost variances resulting from progress deviations due to either over or under-spending. The approach compares the *planned* amount of work with what has actually been *completed*, at any point in the project. This is used to determine if *cost, schedule, and work accomplished (percent of scope completed),* are progressing as planned.

The implications of this, in the software development domain, is that software project management boils down to the monitoring and control of what are essentially the three sides of the "Iron Triangle" – key project performance data points are collected to answer a basic set of questions related to the current state of the project, as listed in Table 1.

QUESTION	EVM Data Elements
How much work should be done?	Budgeted Cost for Work Schedules (BCWS)
How much work is done?	Budgeted Cost for Work Performed (BCWP)
How much did the "is done" work cost?	Actual Cost of Work Performed (ACWP)
What was the total job supposed to Cost?	Budget at Completion (BAC)
What do we now expect the total job to Cost?	Estimate at Completion (EAC)

Table 1- Key EVM Data Points

After this data is collected it is then used to derive some key project performance indicators, some of which are listed in Table 2 below.

Metric	Symbol	Formula	Description
Percent Complete	%Done	$\frac{BCWP}{BAC}$	Ratio of work accomplished in terms of the total amount of work to do.
Cost Performance Index	CPI	$\frac{BCWP}{ACWP}$	Ratio of work accomplished against money spent (Efficiency Rating: Work Done for Resources Expended)
To Complete Performance Index	TCPI	$\frac{BAC - BCWP}{EAC - ACWP}$	Ratio of work remaining against funds remaining (Efficiency which must be achieved to complete the remaining work with the expected remaining money)
Schedule Performance Index	SPI	$\frac{BCWP}{BCWS}$	Ratio of work accomplished against what should have been completed. (Efficiency Rating: Work done as compared to what should have been done)
Estimate At Completion	EAC	ETC + ACWP	Calculation of the estimate to complete plus the money already spent. I.e. how much do we expect the total project to cost
Estimate To Complete	ETC	$\frac{BAC - BCWP}{CPI}$	Calculation of the budgeted work remaining against the performance factor. I.e. how much more will the project cost than planned?

Table 2- EVM Key Data Analysis Calculations[2]

These indicator metrics are inspected at regular intervals to spot trends, take corrective actions, and ensure that a project is on track. Figure 4 below illustrates the tracking of EVM metrics over time.

[2] www.acq.osd.mil/pm

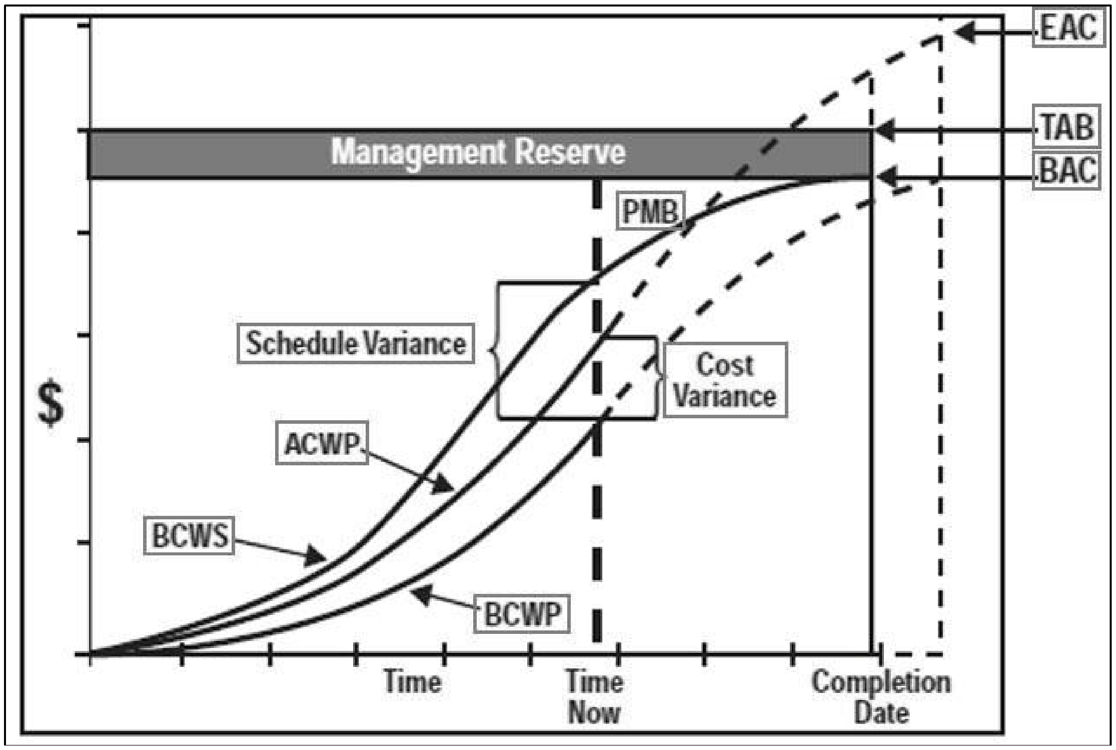


Figure 4- Example Earned Value Chart[3]

Organizations that have traditionally practiced this EVM style of project management have paired it with a "Waterfall" approach to engineering and CMM-inspired processes. While EVM might at first seem complicated or hard to grasp, it is essentially a formal technique for understanding:

- Is work being accomplished as planned?
- Is it costing as planned?
- What is the remaining work likely to cost?
- When will the project be finished?

When management periodically takes time to focus on these questions and takes corrective actions to steer the project back on course when there is a deviation from the plan, EVM is reported to provide a powerful mechanism for large-scale project control. However, in the software engineering realm there are two big problems with EVM:

1) Controlling a project to-plan requires a full up-front plan. This means that the scope of the project must be fully understood, specified and planned up front. Any later changes in direction are regarded as scope changes and must go through a formal contractual change process. The plan cannot evolve and change as new information or insight is gained. Re-planning ("reprogramming" in project management speak) is a difficult affair in EVM. In other words, a rigid plan is not an Agile plan and thus takes away from the project's potential for "nimbleness".

[3] Source: http://acc.dau.mil

2) EVM milestones are arguably arbitrary. For example, EVM claims a significant portion of "earned value" on a development project after the requirements and the design phases are completed. A project can thus claim to be at 50% completion even though no code has yet been produced. On the other hand, as we will discuss in later chapters, Agile projects track progress by feature: project percent-complete is more accurately tracked based on number of features completed. This is also a more valuable customer proposition, as 50% complete means that they can potentially receive a software delivery with 50% of the functionality already in it.

Agile project management techniques are by comparison light-weight, but focus on some of the same project performance questions (is the work progressing as planned? Is it costing as planned?) using mechanisms such as 'sprint burn down charts' (monitors how much work is remaining in one sprint) and 'team velocity' (how much effort a team can handle in one sprint).

We feel that EVM and Agile project management are not at odds and in fact can be mostly complementary. What would be needed, for Agile and EVM to coexist is Agility on the management side. Management must be able to practice what is known as "rolling wave planning" on a near-continuous basis. For example: plan only a month or so ahead of time, thus allowing for a greater degree of flexibility in the project plan (or more specifically, the Performance Management Baseline - PMB) against which performance is measured.

2.2 Waterfall and Big Design Up Front (BDUF)

The origins of the so-called "Waterfall" approach can be traced back to Winston Royce, a director at Lockheed Software Technology in the 1970s, and his now-famous paper from the 1970 IEEE proceedings, entitled "Managing the Development of Large Software Systems." Although the term "waterfall" was not used in this paper, it was the first to describe the process that has come to be called "Waterfall", due to its graphical representation depicting workflow trickling from one stage of the process to the next, reminiscent of an actual waterfall. Figure 5 shows Royce's original depiction of the Waterfall process.

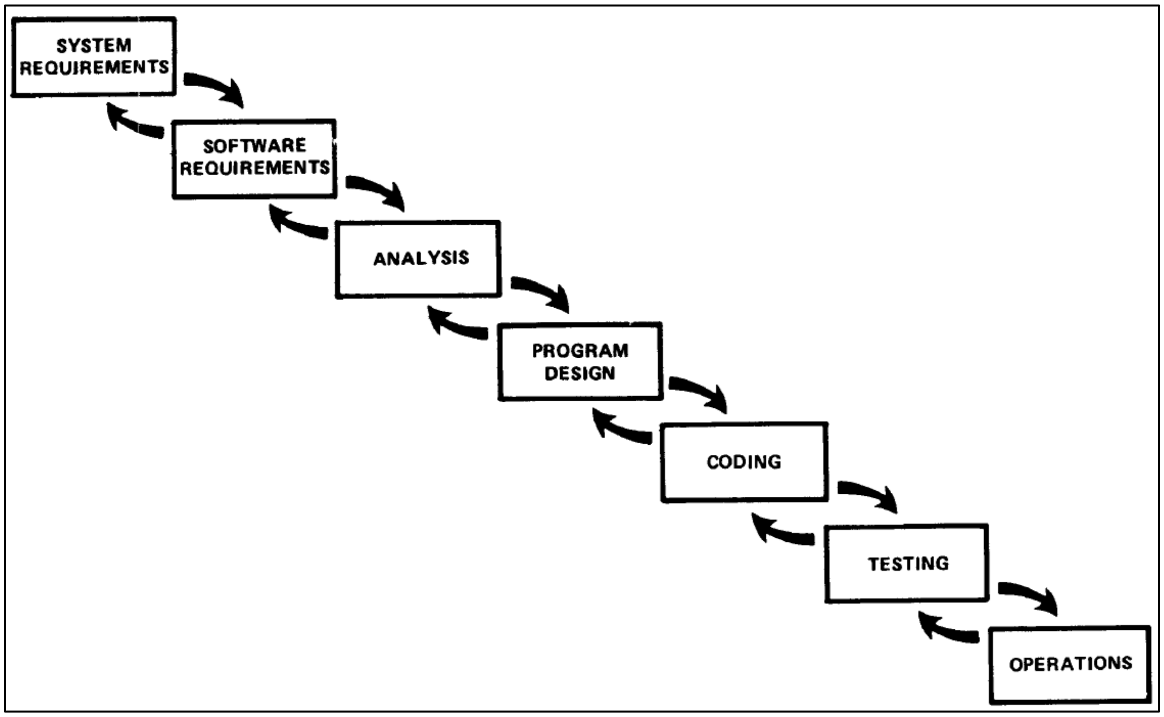


Figure 5 - Winston Royce's Waterfall

The Waterfall as an engineering approach is said to have probably first been applied on a large-scale project by IBM during the development of the System/360 operating system in the 1960s. (Cusumano & Smith 1995)

In a 2004 interview with Fred Brooks, director of the IBM System/360 project, he was asked "Have we learned anything about software development in 40 years?" His response was: "We've learned that the waterfall model of development has to be abandoned, and it hasn't been yet. We need to adopt the spiral model …" We discuss the spiral model in section 3.3.2.

The waterfall process has the advantage of focusing early on complete requirements specification (getting it right the first time,) and since most software projects are said to suffer from poor requirements definition, this approach puts an emphasis on requirements. However, in reality requirements changes are unavoidable ("we might as well embrace it," according to Agile practitioners) – The problem with the waterfall is that the cost of change is prohibitive, and very slow. This is because at the end of each phase, the outputs are certified (via formal inspection and verification) and become the inputs for the next phase. Team members are not supposed to change outputs that the project has already certified. (Cusumano & Smith 1995)

Much of the debate in software engineering related to Agile development frames the discussion in terms of "Waterfall vs. Agile." This is not exactly an appropriate comparison. Waterfall simply describes a "stage-wise" model of development in stages (e.g. planning, analysis, implementation, test, deployment) whereas Agile describes project team performance and use of "agile methods" as we will elaborate in section 3. In other words, a software team can be Agile within a waterfall development model, and a Waterfall iteration can be used within an Agile project – However for simplicity in the context of this research we will continue to contrast Agile with Waterfall, with the term "Waterfall" being a placeholder for "traditional, stage-wise, complete design up-front, single-pass waterfall software engineering approach."

2.3 The Software Capability Maturity Model (SW-CMM)

To deal with the "software crisis" in the 1980s, the US Air Force prompted the Software Engineering Institute (SEI) to develop a method for selecting software contractors. A study of past projects showed that they often failed for non-technical reasons such as poor configuration management, and poor process control.

In essence, the SEI took the ideas of "software lifecycle" and along with the ideas of quality maturity levels, and developed the SW-CMM, or simply CMM, which rates a software organization's maturity level. In CMM, maturity is measured on a scale of 1 to 5. Rather than enumerating and describing each level, suffice it to say that maturity ranges from level 1, or "initial," which applies to the least mature organization (e.g. a startup with two developers) up to level 5, "optimizing," which applies to the most mature of organizations, whose processes are standardized, repeatable, predictable, and optimizing.

There is a set of Key Process Areas (KPAs) that an organization must address in order to rate at each level. Each KPA is met when the organization has a defined process for it, and is committed, and able to follow that process, all while measuring, analyzing and verifying implementation of said process. For example, an organization must have defined practices for Software Configuration Management for a Level 2 or higher rating.

The implication here is that with a higher maturity levels comes a higher software process performance. From a management perspective, predictable software development means better performance on cost and productivity.

The CMM represents a good set of software "common sense". One of the good things about it is that it guides an organization on **what** process areas to address, but does not dictate **how** this must be done. This sometimes leads organizations down a slippery slope of process overload, where an exorbitant amount of time and money is spent developing, maintaining, and deploying processes aimed complying with the CMM KPAs.

In 2002, the SEI's Capability Maturity Model Integration (CMMI) replaced the CMM. The difference, at a high level, is that the CMMI added new "process areas" to the CMM (e.g. Measurement Analysis,) and better integrated other non-software development process in of product development (e.g. Systems Engineering activities.)

Practices added in CMMI models are improvements and enhancements to the SW-CMM. Many of the new practices in CMMI models are already being implemented by organizations that have successfully implemented processes based on the improvement spirit of SW-CMM best practices (SEI 2002). For the purposes of this research we will use the terms "CMM" and "CMMI" interchangeably to mean "process improvement model that defines key processes for product development enterprises."

As a developer I experienced the process overhaul leading to an initial assessment of a CMM level 3 in the late nineties, and later as a software development manager I saw our organization mature to a CMMI level 5. Today we have a "Common Process Architecture," a complex framework of process Work Instructions and measurement capabilities designed as our implementation of the CMMI.

The problem with "mature" organizations, perhaps, is that we become encumbered with the cost and effort of managing processes, collecting, analyzing, and reporting on metrics (in fact the CMMI introduced a whole process area for Measurement and Analysis.) "Optimizing" means continuous improvement: so we do things like six sigma projects to improve and optimize processes. One must wonder whether the software community that the authors of the original CMM document purport to have come to "broad consensus" with really represents the whole of the software community, because we cannot find examples of large commercial software firms such as Google or Microsoft adopting CMMI or boasting high CMMI ratings.

Since 2002 the Defense Department has mandated that contractors responding to Requests for proposal (RFP) show that they have implemented CMMI practices. Other branches of the Federal Government also have begun to require a minimum CMMI maturity level. Often federal RFPs specify a minimum of CMMI level 3. It is therefore no surprise that the contractor community has embraced the CMM (and subsequently the CMMI). It is reported to have worked well for many large, multi-year programs with stable requirements, funding, and staffing.

Traditional criticisms of the CMM are that it imposes too much process on firms, making them less productive and efficient. It is said that only organizations of a certain scale can afford the resources required to control and manage all of the KPAs at higher maturity levels. But again, the CMM tells us what processes areas to address, not how to address them (although best practices are suggested.) Problems arise when firms design their processes ineffectively.

We feel the need to mention the CMM in this research because, in industry, the case is often presented as "Agile vs. CMM". In fact, CMM is *not* at odds with Agile. Au contraire they could be very complementary. The CMMI version 1.3 was released in November of 2010, adding support for Agile. Process areas in the CMMI were annotated to explain how to interpret them in the context of agile practices. There are even several published examples of "Agile CMMI Success Stories", for example the CollabNet project at a large investment banking firm[4]. Again, the CMM only guides an organization on **what** process areas to address, but does not dictate **how** this must be done. Organizations only get into trouble when they over-engineer their processes, making them so cumbersome and time consuming that, as a result, projects lose their potential for agility and responsiveness because they are burdened by the weight of their engineering processes.

2.4 Agile Software Development

In the 1990s, as the large software firms were "maturing" along the CMM dimension, and coinciding with the internet boom and massive growth in the commercial software industry, a parallel movement was taking place: lightweight software development methods, so-called "Agile" methods were evolving, focusing on "soft" factors such as cross-functional teams, customer involvement, and face-to-face communication, collaboration, and creativity. Agile approaches emphasize rapidly building working software, rather than spending a lot of time writing specifications up front. Agile also is geared towards incremental delivery and frequent iteration, with continuous customer input along the way.

One of the early examples of agile methodologies is eXtreme Programming (XP,) which appeared in the mid-nineties. *Extreme Programming emphasizes teamwork. Managers, customers, and developers are all equal partners in a collaborative team. It implements a simple, yet effective environment enabling teams to become highly productive[5].* It preaches such practices as pair-programming, automated unit-testing and incremental development.

In the dot-com era, time-to-market became critical for web-based startups, as the business became an extremely high-velocity environment. The ecosystem of users, needs, and technologies were changing at a break-neck speed. One of the main realizations that came with this was that "change is inevitable, so we might as well embrace it." This goes against the traditional Change-Management approach of locking down requirements and specifications at the outset of a project; however it was the reality for most software developers in the commercial realm.

In February 2001, a group of experienced software professionals, representing practices ranging from Extreme Programming to SCRUM, and others sympathetic to the need for an alternative to heavyweight software development processes, convened in Utah. They formed the Agile Alliance and what emerged was the "Agile Manifesto"[6], which put forth a declaration of the following core **values**:

[4] http://www.open.collab.net/media/pdfs/AgileCMMI_CollabNet.pdf
[5] http://extremeprogramming.org
[6] http://agilemanifesto.org

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

In addition to these values, the "**Twelve Principles of Agile Software**" were also declared:

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
Business people and developers must work together daily throughout the project.
Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
Working software is the primary measure of progress.
Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
Continuous attention to technical excellence and good design enhances agility.
Simplicity--the art of maximizing the amount of work not done--is essential.
The best architectures, requirements, and designs emerge from self-organizing teams.
At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Table 3 - Principles Behind the Agile Manifesto[7]

[7] http://agilemanifesto.org/principles.html

Commercial software companies started adopting agile practices. Today there are dozens of purportedly agile methodologies. Yet it is hard to point to a single document, framework, or process description to find out exactly what defines Agile.

2.5 The Case for Agile in Government Software Development

A common attitude in the software industry has been that Agile only works for small-scale projects with small experienced teams, and that CMM/Waterfall is better suited projects of larger scale. However, the waterfall development life cycle, based on an assumption of a relatively stable business environment, becomes overwhelmed by high change (Highsmith 2002b). High change is exhibited in today's environment: whether it is to support the war efforts in the 2000s, to address rapidly-evolving cyber-security threats, or to compete in an ever-more net-centric world: the demands of the software customer today are such that capabilities are needed on shorter time horizons, quality expected out of the gate, and the software producers are expected to be nimble enough to mirror the fast-paced changes in their customer's needs and environment changes.

Taking inspiration from the world of biology, Charles Fine argues that each industry has its own evolutionary life cycle (or "clockspeed"), measured by the rate at which it introduces new products, processes, and organizational structures (Fine 1996). The government systems software market had historically been a slow-to-medium-clockspeed environment, with software being only a component within large multi-year projects. It is now becoming a fast-clockspeed environment, with demand for quick software delivery to run on existing networks, infrastructure, and systems. Fine argues that firms must adapt to the changing environment to maintain any competitive advantage. Government software contractors must then adapt to survive.

In 2010, the Under Secretary of Defense for Acquisition, Technology and Logistics, Dr. Ashton Carter (now deputy Secretary of Defense) issued a memo entitled "Better Buying Power: Guidance for Obtaining Greater Efficiency and Productivity in Defense Spending "– He was on a mission to rein in cost overruns and spending as part of an overall plan by Defense Secretary Robert Gates to cut $100 billion from the Pentagon budget over the subsequent five years. Carter also directed acquisition managers to award more services contracts to small businesses because they provide Defense with "an important degree of agility and innovation," with lower overhead costs (Berwin 2010).

The big government software contractors took note… The Pentagon budget is shrinking and they now will increasingly have to compete with small/nimble firms for contracts. This is perhaps the single driving force behind why these companies are taking a serious look at disrupting their mature product development practices by incorporating Agile software development as methods that are practiced in the commercial world, including in today's dominant "born-on-the-web" software giants like Google, Facebook, and Amazon. A Forrester Research survey of 1000+ IT professionals in 2009, followed by one in 2010, (see Figure 6 in section 3.2) shows a decline in the use of "Traditional" development methodologies, and a rise in adoption of Agile.

3. Mapping the Genome of Agile Development

The most important thing to know about Agile methods or processes is that there is no such thing. There are only Agile teams. The processes we describe as Agile are environments for a team to learn how to be Agile.

- *Kent Beck, creator of eXtreme Programming*

3.1 Defining the term “Agile”

So far in this paper we have been using the words “agile” and “Agile” without pausing to ponder the meaning of or explain the context within with they are being used. At the outset of this research, I had a very narrow view of what Agile meant. This was based on my experiences with XP and Scrum. Subsequently, through research and discussion with software professionals from various software industry domains at M.I.T., I quickly realized that there is no consensus on the definition of Agile and that different mental models exist pertaining to how agile practices fit within the software engineering discipline.

The American Heritage Dictionary defines “agile” as: *Characterized by quickness, lightness, and ease of movement; nimble*. These are the characteristics that we as software project managers desire to imbue into our projects, and that we as software developers want to see in our development practices. We want to be able to quickly produce high-quality software and respond in near-real time to changes in customer needs and requirements, while reaping benefits in terms of cost and schedule.

Another popular word in industry that has come to represent this concept is “lean.” The terms Lean Software Engineering, Lean Software Development, and other similar constructs are also being used in conjunction with or en lieu of Agile. The lean concept finds its roots in manufacturing and supply chain management, tracing back to the Toyota Production System. Lean relates to Agile in that the end goal is to produce product (software or other) faster, cheaper, and more efficiently. Lean principles revolve around process improvement, such as improving efficiency (eliminating waste), Just-In- Time (JIT) engineering, rapid delivery, and team empowerment. For our purposes, Lean is the set of "tools" that assist in the identification and steady elimination of “muda” (a Japanese word meaning an activity that is wasteful and doesn't add value or is unproductive). As waste is eliminated, quality improves while production time and cost are reduced (Gershon 2011). As will be presented later in this research, Agile employs Lean tools and principles to deliver value.

Hereafter, when we refer to “agile software development” and “agile development methods” we are referring to practices geared towards improving the engineering effort of software production. When we refer to “agile project management” we are referring to the management of agile software development projects, and not an agile approach to project management in general. Finally, there is no definitive definition for the capitalized term “Agile” as it is used in various contexts to mean various things. To avoid semantic complications, in this research we use Agile an umbrella term symbolizing the general movement towards agile methods in software engineering and its management.

3.2 A Brief Review of Agile Methodologies

This research aims to produce a System Dynamics model of Agile software projects. In order to model Agile, we must first understand the essence of agility: What makes software development agile? In 2009 and 2010 Forrester Inc. surveyed 1298 and 1093 software professionals respectively, asking them to select the methodology that most closely reflected their development process. As shown in Figure 6, respondents pointed to several popular development methodologies, all of which fall under the Agile umbrella (West et al. 2011). In 2010 almost 40% of all responses named one of the "agile family" of methodologies.

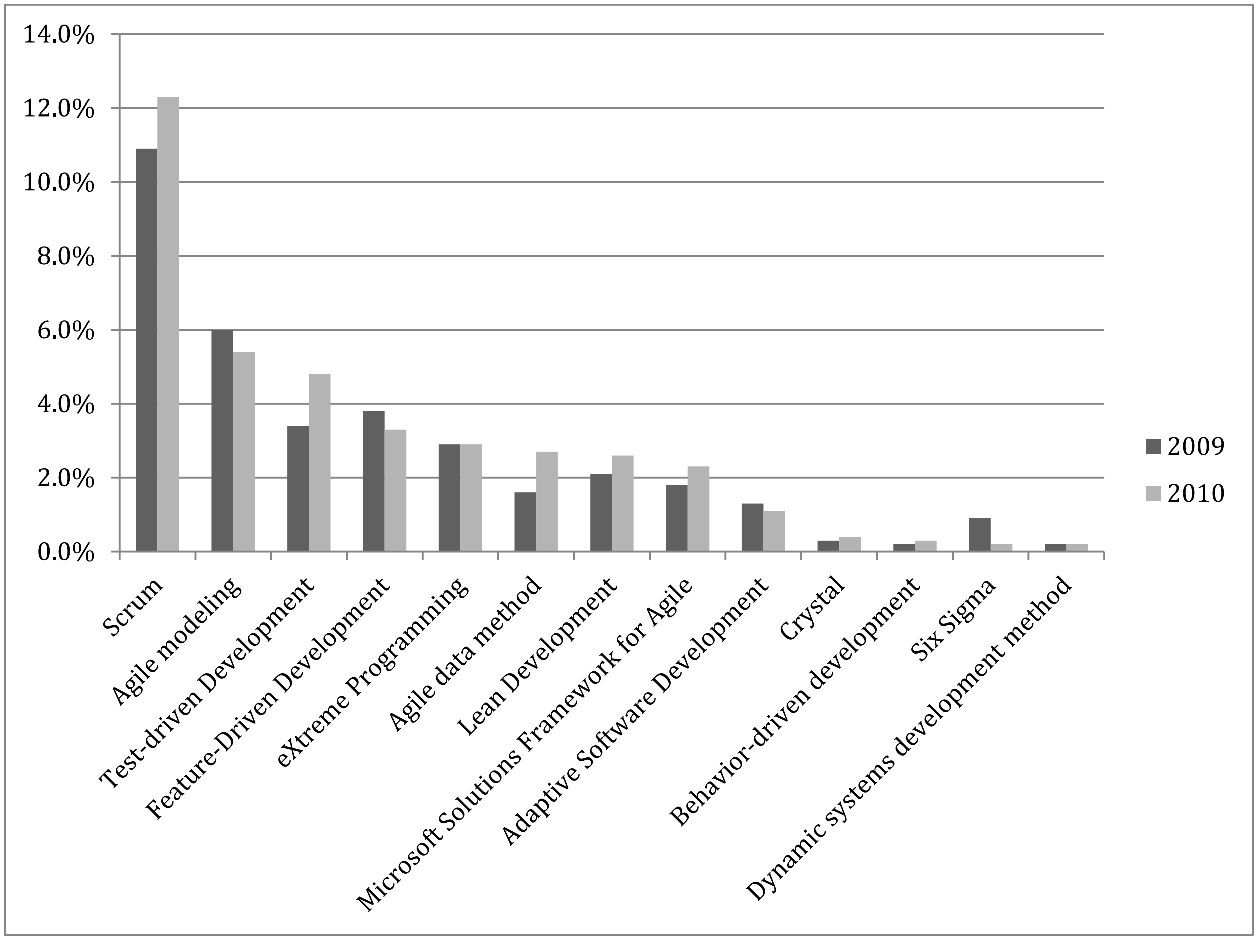


Figure 6 – Forrester Inc. Survey of Agile Methodologies Practiced

The respondents identified some of the most in-use Agile methodologies, with Scrum being the most popular. In the following section we take a very brief look at some of these methodologies.

3.2.1 Scrum

Scrum's origins can be traced back to an article that appeared in the January 1986 Harvard Business Review, entitled "The New New Product Development Game". It contrasts Waterfall-like practices at the National Aeronautics and Space Administration (NASA) to novel approaches at companies like 3M, Fuji-Xerox, Honda, and Cannon. The Waterfall approach is likened to a relay-race, while approaches at successful companies were portrayed as being more akin to rugby teams "moving the scrum downfield" (Takeuchi & Nonaka 1984). Over time, what they called the "rugby approach" later morphed into "Scrum".

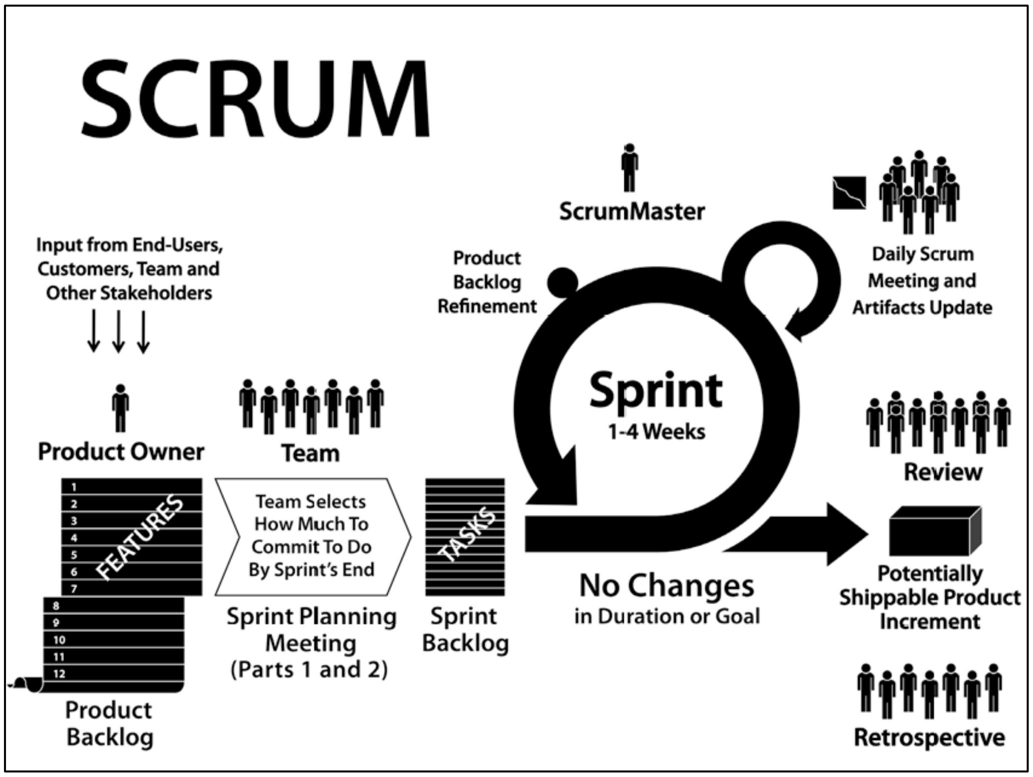

SCRUM
Input from End-Users, Customers, Team and Other Stakeholders
Product Owner
Team
ScrumMaster
Product Backlog Refinement
Daily Scrum Meeting and Artifacts Update
Sprint
1-4 Weeks
Review
FEATURES
Team Selects How Much To Commit To Do By Sprint's End
TASKS
Sprint Planning Meeting (Parts 1 and 2)
Sprint Backlog
No Changes
in Duration or Goal
Potentially Shippable Product Increment
Product Backlog
Retrospective

Figure 7 - Representation of the Scrum Process (Deemer et al. 2009)

Scrum is an iterative, incremental framework for projects and product or application development. It structures development in cycles of work called Sprints (Deemer et al. 2009). Each sprint is typically a three to four week long development cycle wherein self-organizing teams prioritize and select a subset of the features of the software to implement. At the end of each sprint, completed features are demonstrated to the customer or user, whose feedback is incorporated into the software in later sprints. This methodology also calls for short daily stand-up meetings, also known as daily scrums, where team members exchange information on progress, next tasks, and surface issues and roadblocks so that they are dealt with quickly. Each sprint culminates with a set of completed software features that are “potentially shippable.” Figure 7 above is a simplified depiction of this process.

3.2.2 Extreme Programming

Extreme Programming, also referred to as XP, is a lightweight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements (Beck 1999). Like Scrum, it emphasizes iterative and incremental delivery of small releases, as depicted in the XP project flowchart in Figure 8.

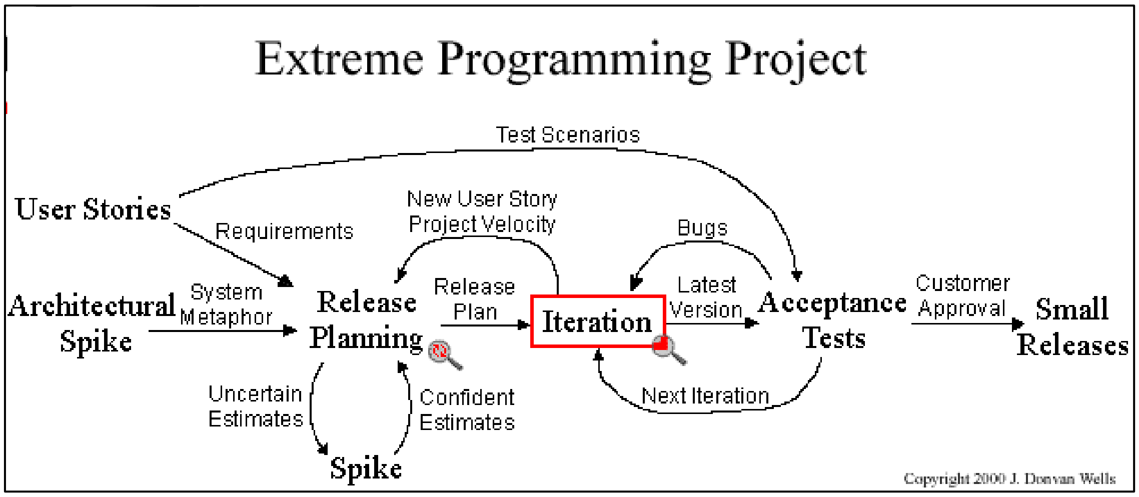


Figure 8 - The XP Development Flowchart

The development practices of XP teams are governed by the following rules:

- *The Planning Game.* Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes the plan, update the plan.
- *Small releases*. Put a simple system into production quickly, then release new versions on a very short cycle.
- *Metaphor*. Guide all development with a simple shared story of how the whole system works.
- *Simple design*. The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.
- *Testing*. Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write tests demonstrating that features are finished.
- *Refactoring*. Programmers restructure the system without changing its behavior to remove duplication, improve communication, simplify, or add flexibility.
- *Pair programming*. All production code is written with two programmers at one machine.
- *Collective ownership*. Anyone can change any code anywhere in the system at any time.
- *Continuous integration*. Integrate and build the system many times a day, every time a task is completed.

- *40 hour week*. Work no more than 40 hours a week as a rule. Never work overtime a second week in a row.
- *On-site customer*. Include a real, live user on the team, available full-time to answer questions.
- *Coding standards*. Programmers write all code in accordance with rules emphasizing communication through the code.

XP was perhaps the first methodology, in the late 1990s, which called for automated unit testing of code, and is renowned for also being the first to call for the still-to-this-day controversial practice of pair programming.

3.2.3 Test Driven Development

Inspired by the "test first" philosophy from XP, Test Driven Development (TDD) starts the development process by coding automated test cases for the software features that are to be produced. Then, production code is developed iteratively until all the tests are passed. After each cycle (iteration), all of the tests are re-run to ensure that new functionality is well integrated. Refactoring is performed to maintain quality and remove duplication in both production and test code. Figure 9 illustrates this process as practiced by a software development group at IBM (Maximilien & Williams 2003).

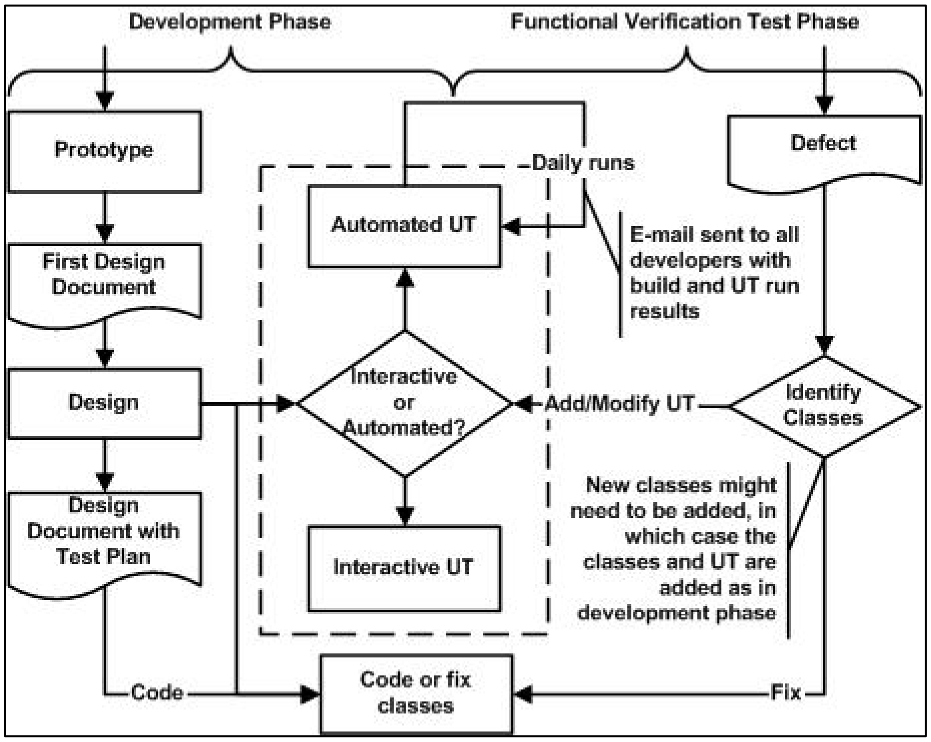

Development Phase
Functional Verification Test Phase
Prototype
First Design Document
Design
Design Document with Test Plan
Daily runs
Automated UT
Interactive or Automated?
Interactive UT
Defect
E-mail sent to all developers with build and UT run results
Add/Modify UT
Identify Classes
New classes might need to be added, in which case the classes and UT are added as in development phase
Code
Code or fix classes
Fix

Figure 9 - Test Driven Development Process at IBM (UT = Unit Test)

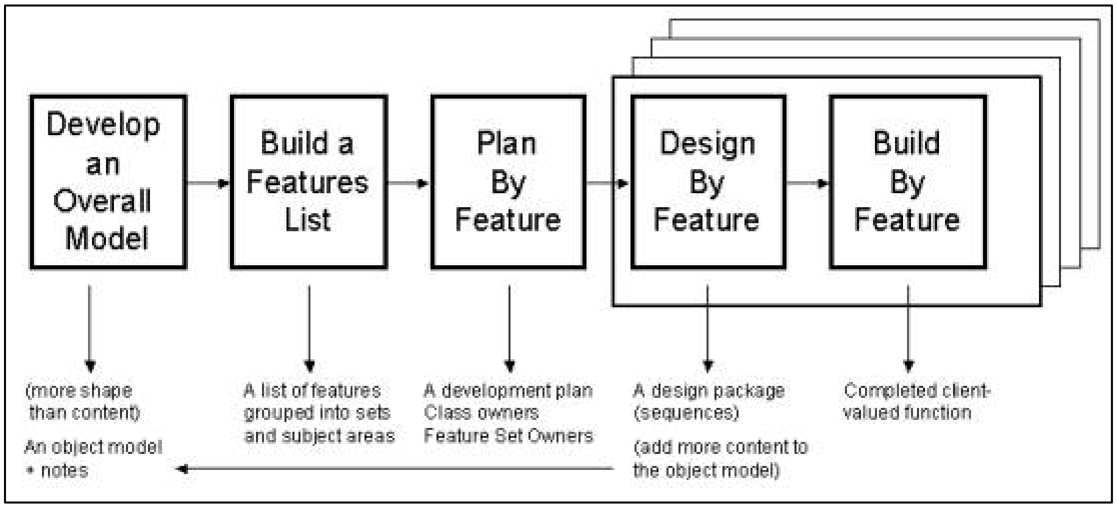
TDD's test-first approach is reported to have a significant impact on defect prevention, but more importantly it influences the design of the software. Since developers must focus on the interfaces of their modules (this helps in developing modules to pass the unit tests) it means that they must employ a type of "Design by Contract" (often referred to as “DbC”) approach to software development. Test-first code tends to be more cohesive and less coupled than code in which testing isn’t part of the intimate coding cycle (Beck 2001). In a nutshell, Kent Beck, the creator of XP and TDD, lists the benefits of a test-first approach as follows:

- Encourages you to be explicit about the scope of the implementation,
- Helps separate logical design from physical design from implementation,
- Grows your confidence in the correct functioning of the system as the system grows.
- Simplifies your designs.

3.2.4 Feature Driven Development

Feature Driven Development (FDD) is described as having “just enough process to ensure scalability and repeatability and encourage creativity and innovation all along the way” (Highsmith 2002a). As shown in Figure 10, FDD breaks the system into feature sets, and iterates to produce incremental client-valued pieces of functionality. FDD can be summarized by its eight “best practices”:

- *Domain object modeling*: Since FDD was developed originally in 1997 for a Java language based project, it is tailored to an Object-Oriented (OO) approach. FDD calls for building class diagrams to capture the attributes and relationships between the significant objects in the problem space.
- *Developing By Feature:* The system is broken up into a set of features which can be developed incrementally. In FDD, a feature is a small, client valued function that can be implemented in two weeks (Goyal 2007).
- *Individual class ownership*: Unlike XP, which calls for “collective code ownership”, FDD asks that each class (a unit of code in OO programming) is assigned to an individual who is ultimately responsible for it.
- *Feature teams*: Features are developed by teams comprised of feature owners and a combination of the class owners needed to implement the given feature.
- *Inspections*: formal code reviews are held to prevent defects and ensure quality.
- *Regular builds*: Allows early detection of integration problems, and makes sure there is always a current build available to demo to the customer.
- *Configuration management*: The use of source control and version tracking.
- *Reporting and visibility of results*: Progress for each feature is based on the completion of development milestones (e.g. Design completion, Design Inspection Completion, etc.) Progress of the feature sets is regularly reported.



Develop an Overall Model
Build a Features List
Plan By Feature
Design By Feature
Build By Feature
(more shape than content)
An object model + notes
A list of features grouped into sets and subject areas
A development plan
Class owners
Feature Set Owners
A design package (sequences)
(add more content to the object model)
Completed client-valued function

Figure 10 - Stages of Feature Driven Development[8]

FDD is similar to the "agile modeling" methodology since it relies on a UML (Unified Modeling Language) model of the system.

3.2.5 Crystal Methods

On a quest to develop an effective software methodology, Alistair Cockburn interviewed and studied project teams for 10 years. He found that "people-centric methodologies" do better than "process-centric" methodologies, and that one must choose and tailor the methodology to the team and the assignment - no methodology fits all projects. (Cockburn 2004)

The result is Crystal, which is actually a family of methods, rather than a single methodology, developed to address the variability between projects. Projects are sized along two dimensions: team size, and program criticality. A version of Crystal is subsequently chosen and adapted to the specifics of the project. See Figure 11.

[8] http://www.step-10.com/SoftwareProcess/FeatureDrivenDevelopment/FDDProcesses.html

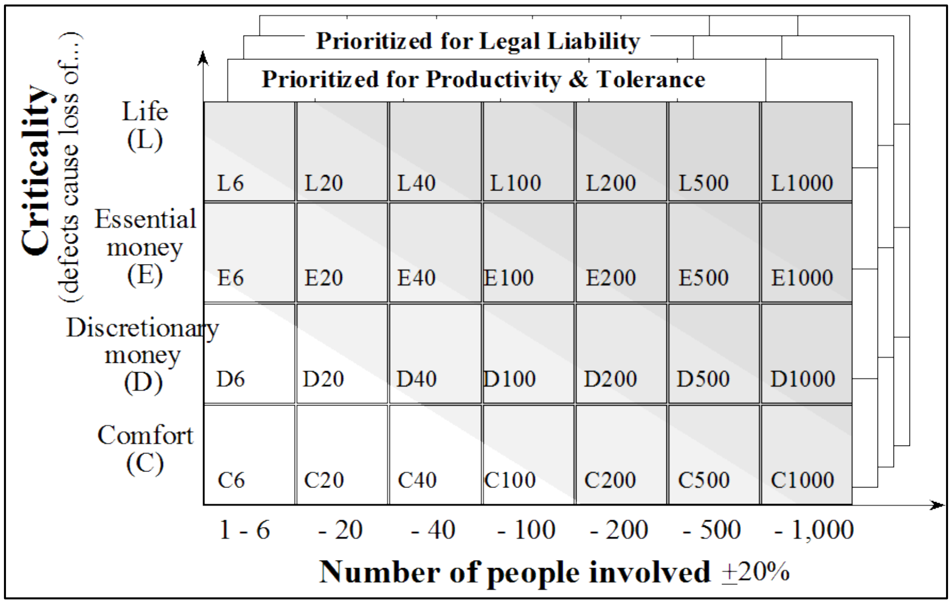


Figure 11 - The Crystal family of methods.[9]

Team Size accounts for the fact that as a team gets larger communication costs rise and face to face communication becomes less effective. More management and coordination become required. Criticality on the other hand measures the system's "potential for causing damage" ranging from "loss of life" to "loss of comfort." The combination of team size and the criticality directs a given development effort towards a corresponding Crystal methodology. There are only two rules that govern the practice of the Crystal family of methods.

1) Incremental cycles cannot exceed four months.
2) Reflection workshops must be held after every delivery so that the methodology is self-adapting.

Crystal "focuses on people, interaction, community, skills, talents, and communication as first order effects on performance. Process remains important, but secondary" (Highsmith 2002a)

[9] http://alistair.cockburn.us/Crystal+light+methods

3.3 The Agile Genome

After review and analysis of many Agile methodologies, some of which were described in section 3.2, we come to find that they all share common characteristics. The project teams that employ these methodologies in effect practice a variety of remarkably similar "agile techniques."

When seeking to identify the basic set of common attributes that these Agile methodologies share, we can start to develop a set of characteristics that we can call "the Genome of Agile" – We have distilled these into the following seven genes.

3.3.1 Story/Feature Driven

A principle of the Agile Alliance is that "Working software is the primary measure of progress." Most Agile teams break up their projects into manageable sets of "features", "stories", "use cases", or "capabilities," rather than architecting the complete system up-front as is done in classic Big Design Up Front (BDUF) approaches. The terminology differs across the various methodologies, but the concept is the same. Feature Driven Development is perhaps the most obvious example of such a methodology, as it involves building a feature list, then planning, designing, and implementing the software feature by feature. Note that not all features are equal in size, complexity, or priority. In most agile methodologies, features are sized or weighted depending on an estimate of the amount of effort required to implement the feature. Feature planning activities must also take into account feature inter-dependencies, and plan accordingly.

From a management perspective, the implication of using this approach is that management can have a concrete measure of progress by-feature (i.e. 9 out of 10 features implemented means 90% complete, assuming all features weighted equally.) This differs from traditional EVM-like measures of progress, based on arguably arbitrary milestones, where for example, the completion of the design phase translates to claiming forty percent of the project complete. One of problems with using a waterfall method paired with EVM management methodology is that a project can report to be at 90% completion, yet still have no functioning software.

Another advantage of the feature-driven approach is that, as features are developed and integrated into the software, they become available for early customer demonstrations as well as early integration and test activities – all of which help reduce uncertainty and detect defects early in the development cycle, as opposed to waiting for a complete integrated build.

The downside of a featured approach is that over time, the software's architecture and code start to exhibit signs of having "high coupling" and "low cohesion," making it harder (and more costly) to maintain and evolve. Coupling refers to the degree to which software modules, components, etc., depend on each other – in a system with high coupling there is a high degree of dependency, meaning that changes to one software element is likely to have ripple effects and impact on the behavior of other elements. Cohesion is a measure of how strongly related the responsibilities of a single software module or component are – low cohesion is an indicator of lack of structure in the system. (for example, a software library or component that provides a large set of completely unrelated functions or services is said to exhibit low cohesion, whereas one that only specifically provides string manipulation functions is said to be highly cohesive.) The segmentation of a system by feature can lead to "high coupling" and "low cohesion." Higher coupling and lower cohesions means that there are a lot of software interdependencies; changes to one area of the software will have more impact on other parts of the system, and thus makes future changes more costly and difficult to make. This is why Refactoring (see section 3.3.3) is called for by most feature-driven methods.

3.3.2 Iterative-Incremental

Another principle of the Agile Alliance is to "deliver working software frequently, from a couple of weeks to a couple of months." Development is performed in repeated cycles (iterative) and in portions at a time (incremental.) This allows developers to:

- Take advantage of what was learned during earlier development in later iterations.
- Focus on short term objectives.

In this approach, development will start with a simple implementation of a subset of the software requirements and iteratively enhance the evolving versions until the full system is implemented. With each iteration, design modifications are made and new functional capabilities are added. Most value is derived when iterations are designed such that early tasks help resolve uncertainty later in the project. Rather than one big design phase, one big code phase, then one big test phase, here many iterations are performed, with each iteration consisting of a short design-code-test cycle. (Figure 12)

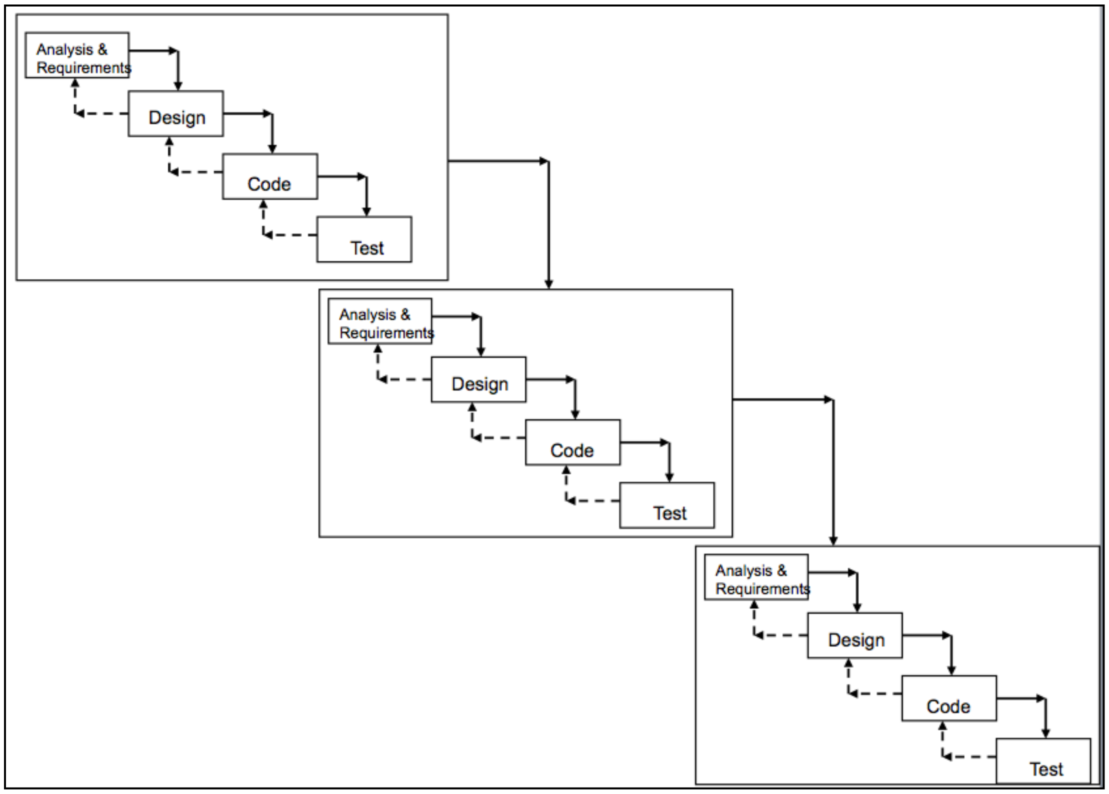


Figure 12- Design-Code-Test iterations (de Weck, & Lyneis 2011)

This "Iterative-Incremental" characteristic combines with the afore-mentioned "feature-driven" gene (see section 3.2.4) to allow the software product to continually evolve as a series of product increments, each one adding more features to the existing software product.

The Iterative-Incremental concept is not novel nor unique to Agile methodologies. There is well documented evidence of extensive Incremental Iterative Development (IID) for major software development efforts dating back to the sixties (Larman & Basili 2003). Software historians seem to agree that Royce's original work on the Waterfall has been misrepresented as calling for a single iteration of the Waterfall, and that he actually proposed several iterations of the process.

In 1994 the DoD's Defense Science Board Task Force on Acquiring Defense Software Commercially, issued a report that stated, "DoD must manage programs using iterative development. Apply evolutionary development with rapid deployment of initial functional capability." (Larman & Basili 2003) The result was a new standard for software acquisition introduced that same year, Mil-Std-498, which stated:

If a system is developed in multiple builds, its requirements may not be fully defined until the final build […] If a system is designed in multiple builds, its design may not be fully defined until the final build.

This allowed projects to start while only needing fully-defined requirements for one build at a time, rather than a full requirements specification for the entire project, allowing later requirements analysis efforts to be informed by work and experience from earlier build – a step in the agile direction. This standard, although later replaced by others, was a first attempt to introduce the concept of lifecycle and incremental delivery to government software projects. It is also an acknowledgement of the fact that Waterfall development and acquisition was problematic, and that the previous DoD standards had a "Waterfall bias" (perceived preference towards a single-pass Waterfall model of development.)

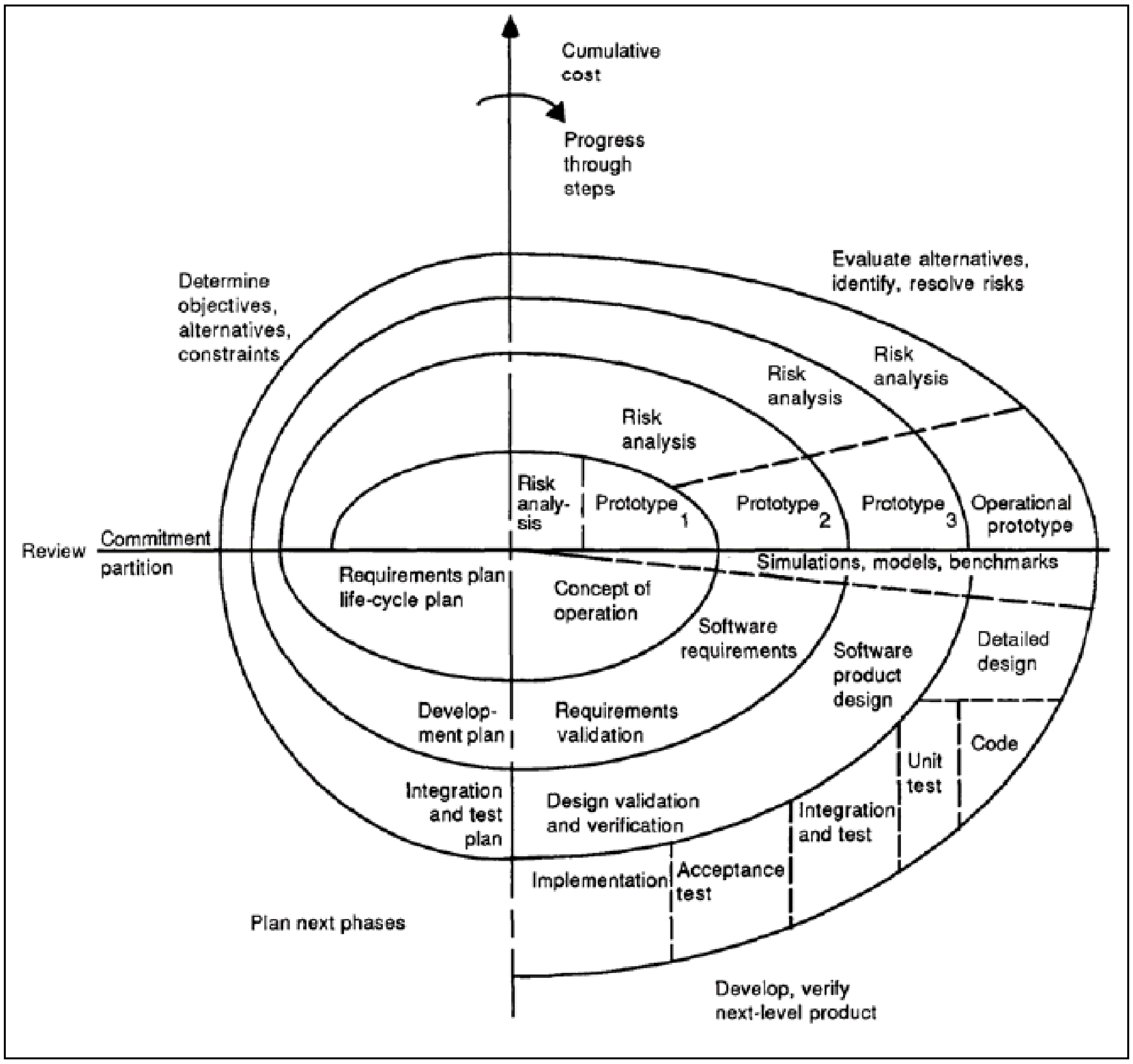


Figure 13 - The Spiral Model of Development (Boehm 1987)

Explicit iteration and incremental development is neither new nor unique to Agile. It first came to the fore-front of the software engineering community's conscience in 1986 with Barry Boehm's "*A Spiral Model of Software Development and Enhancement*". The Spiral model, shown in Figure 13, very simply put, calls for repeated waterfall iterations to build and refine a software product. Early spirals can achieve goals of producing quick-to-market prototypes which can be tested or presented to customers for early feedback, which produces valuable information for later spirals. This approach mitigates project risk and allows requirements to be evolved and refined incrementally, keeping the project agile in that software is built incrementally and that the approach caters to the reality of evolving requirements.

In 2002, the DoD declared "Evolutionary acquisition strategies shall be preferred approach to satisfying operational needs" and "Spiral development shall be the preferred process"[10] The following two acquisition models became the official standards:

- Incremental Development: End-state requirement is known, and requirement will be met over time in several increments
- Spiral Development: End-state requirements are not known at Program Initiation. Requirements for future increments dependent upon technology maturation and user feedback from initial increments.

3.3.2.1 Illustrative Example

To illustrate the difference between Iterative-Incremental Story/Feature-driven approaches vs. a Waterfall/BDUF approach, let us consider the following fictional example: Suppose that we are to develop software for an Automated Teller Machine (ATM). The software is required to allow users to check balances, withdraw/deposit money, and transfer funds between accounts. How would the two approaches differ? Albeit contrived and simplified, this example helps clarify the difference.

3.3.2.2 Waterfall/BDUF

Figure 14 depicts the flow of a classic Waterfall/BDUF approach: In the Analysis phase the requirements for the ATM system may be documented in a "System Requirements" document as the result of discussions, negotiations, analysis, and compromise between the customer agent and the contractor's system engineer or business analyst. This document is usually named something akin to "System/Subsystem Specifications" (SSS). Once the SSS requirements for the system are "locked-in", the next phase "Requirements Specification" can begin.

[10] DoD Instructions 5000.1 and 5000.2

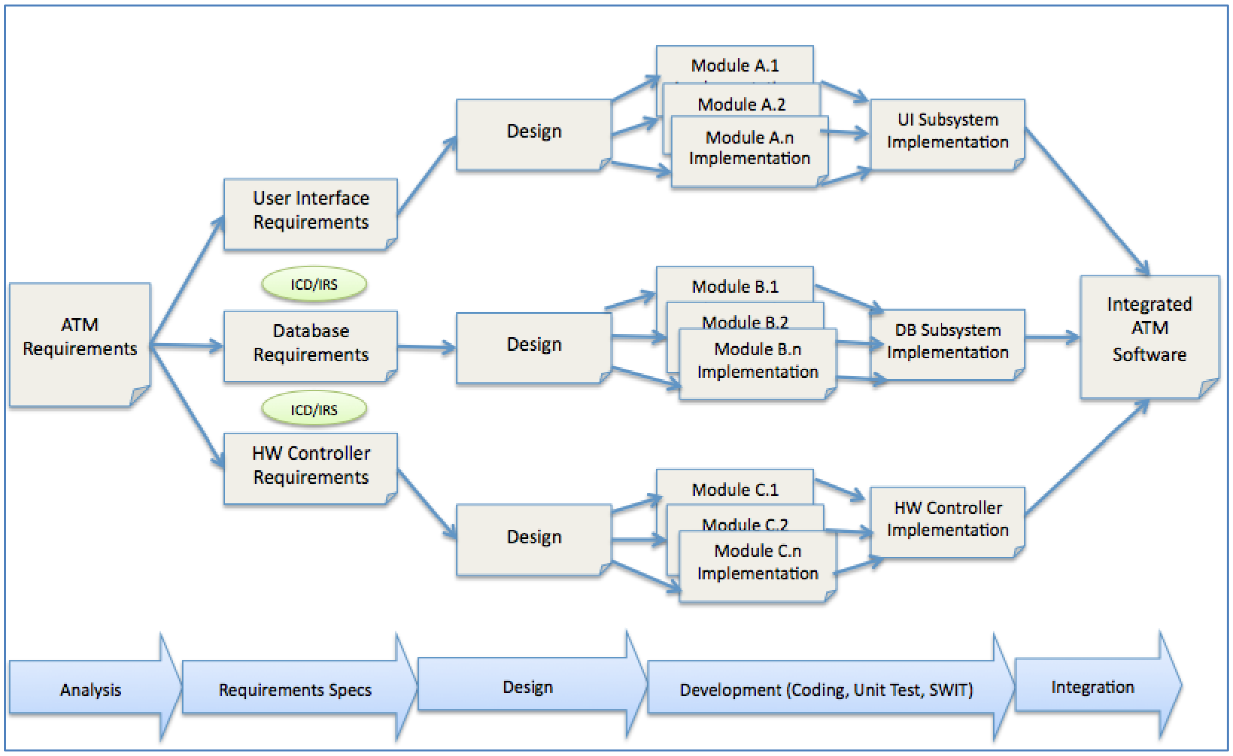

ATM Requirements
User Interface Requirements
ICD/IRS
Database Requirements
ICD/IRS
HW Controller Requirements
Design
Design
Design
Module A.1
Module A.2
Module A.n Implementation
Module B.1
Module B.2
Module B.n Implementation
Module C.1
Module C.2
Module C.n Implementation
UI Subsystem Implementation
DB Subsystem Implementation
HW Controller Implementation
Integrated ATM Software
Analysis
Requirements Specs
Design
Development (Coding, Unit Test, SWIT)
Integration

Figure 14 - Example Waterfall Development Flow

Using a functional decomposition approach, the next levels of requirements are developed: the architecture is produced, which defines three main subsystems, in this ATM example:

- A User Interface subsystem or component, encapsulating the software for interacting with the ATM user.
- A Database component, which is responsible for communicating with the bank's central database and accessing account information.
- A Hardware Controller subsystem for interfacing with the actual ATM hardware.

For each of subsystem, a "Software Requirements Specification" (SRS) document is produced. Also, interfaces between the subsystems/components are specified in some sort of document, named something like "Interface Requirements Document" (IRS) or "Interface Control Document" (ICD). In theory, if each component meets its SRS requirements, and adheres to applicable ICDs, then the system will function as specified in the original SSS.

Next, an individual or team is assigned to the design and development of each subsystem, based on its SRS. A design is produced for each subsystem, followed by the coding of each software module to implement the design. Then, each module is individually tested (Unit Testing.) Once unit testing is completed, the subsystem is tested as a whole, bringing together the individual modules in a "Software Integration and Test" (SWIT) activity. Finally, the components are integrated and the ATM software system is tested as a whole and validated against the original SSS.

3.3.2.3 Story/Feature Driven

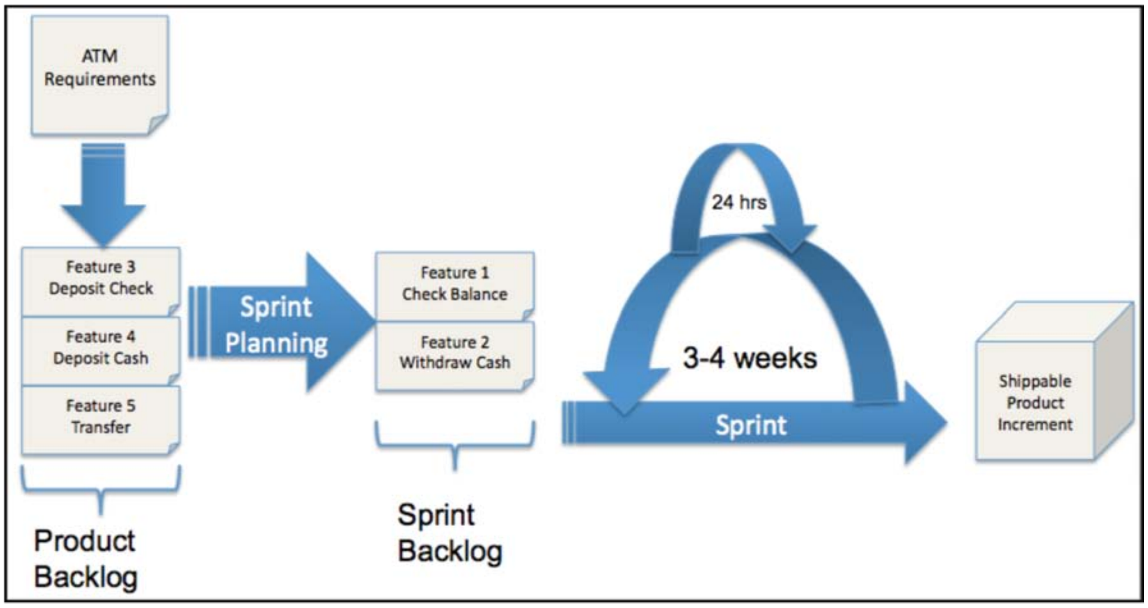


Figure 15 - Feature-Driven Scrum Approach

Figure 5 illustrates development of the same ATM software using a Scrum methodology. Using a Story/Feature driven approach, the ATM system is segmented not into functional components, but rather into a set features corresponding to the system's use cases: In this example, the ATM software's list of features may be: Check Balance, Withdraw Cash, Deposit Check, Deposit Cash, and Transfer Balances. Note that in feature-driven approaches, not all features are equal in size or effort. In Scrum, each feature is sized by "story points", a relative measure of the amount of effort estimated to be needed to implement that feature. For the purposes of this example, let us consider these features equal.

The development team follows by then prioritizing the set of features and starts to develop the software feature-by-feature or a subset of features at a time in short "Sprints". As each sprint is completed, the set of features developed are added as an increment to the software product's baseline producing a "potentially shippable product increment."

3.3.3 Refactoring

An incremental and feature driven approach to the development of software systems can produce sub-optimal architectures compared to a waterfall model, as discussed previously. One of the advantages of BDUF is that the complete up-front design is optimized for a full-featured release. Components are well-integrated and duplication is minimized.

On the other hand refactoring is needed to pay off the "technical/design debt" which accrues over time, especially when incremental and evolutionary design results in a bloated code base, inflexible architecture, duplication, and other undesirable side effects.

The metaphor of technical debt was used by Ward Cunningham (creator of the Wiki) to describe what happens as the complexity and architecture of a software project grow and it becomes more and more difficult to make enhancements. Figure 16 illustrates this concept: as the software product degrades over time, the cost of change increases to the detriment of the customer responsiveness.

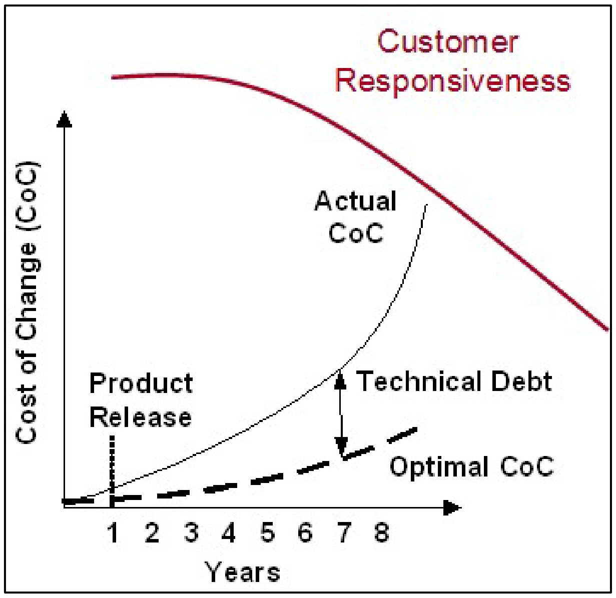


Figure 16 - Technical Debt Curve[11]

This concept of technical debt has become popular as it can be understood by both technical minded and business minded people. Just like credit debt, technical debt accrues interest payments in the form of extra effort that must be made in subsequent development cycles. Management can choose to pay down the principal on this debt by refactoring, and keep the future cost of change as low as possible.

Controlling the cost of change is very important for an Agile project, since the philosophy is to embrace change. Indeed one of the twelve principles of the Agile Manifesto

[11] source: http://jimhighsmith.com

is to *welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage*. (see Table 3 - Principles Behind the Agile Manifesto.) Therefore refactoring is critical to keeping the agile process sustainable through the pay-down of technical debt.

Holistically speaking, one could argue that true Agile is not only about the agility of the development process, or the team, but also about the software product itself. In other words, a "Systems Thinking" perspective would suggest that the software product itself is part of the system under study here, *in addition to* the people, processes, and tools. If the software is constantly refactored to keep it easy to adapt and evolve along with the requirements, needs, or market environment, then the project can truly be agile. Perhaps one of the problems with Agile adoption in large-scale government programs is with attempting to employ it on long-running legacy programs that are already deep in technical debt.

Many agile methodologies (in particular XP) consider refactoring to be a primary development practice. Refactoring has the disadvantage that it takes extra effort and requires changing baseline software without any direct or apparent ROI. A project manager may ask "why are we spending effort re-designing portions of the system unrelated to the next planned set of features?" This is when the technical debt metaphor comes in handy as a tool for communicating the business impact of design and architectural decisions.

The project management may still resist refactoring with: "If we do refactoring, we will have to re-test and re-certify existing baseline functionality, at added effort and cost to the project!" Any change has the potential to reduce the maturity and the stability of the software, requiring regression testing and revalidation of the baseline feature set. This is why it is advantageous to practice refactoring in conjunction with test-heavy practices (e.g. TDD) and Continuous Integration techniques (see section 3.3.7).

An example of a software organization that embraces refactoring as part of its software engineering culture is Google. The following points, taken from Google's Agile Training (Mcfarland 2006), summarize some of the reasons behind their embrace of Refactoring:

- *As code ages, the cost of change goes up*
- *As Google grows, the percentage of code in maintenance mode grows with it*
- *We need to keep code flexible, so we can change it to suit new market conditions quickly*
- *It's not important if all you're trying to do is get new products out. (Smart programmers can develop very good applications without tests, but those applications will be much harder to maintain.)*

A final note on refactoring and technical/design debt is that this phenomenon can be observed at the enterprise level. We observe that much of the work in net-centric architectures, which involves evolving an ecosystem of silo'd systems towards a system-of-systems architecture using technical approaches such as SOA (Service-Oriented

Architecture), can be understood through the lens of technical debt as grand-scale exercises in refactoring.

3.3.4 Micro-Optimizing

This gene represents the adaptive nature of agile processes. We employ the term "Optimizing" because, in most agile methodologies, teams are empowered if not encouraged to modify aspects of the development process or dynamically adapt to changing circumstances. "Micro" is used to indicate that small process adjustments and improvements are made frequently and as needed. For example, the Scrum process requires a "Sprint Retrospective" in between iterations. Likewise, Alistair Cockburn – author of the Crystal methods -- believes that as the project and the people evolve over time, the methodology so too must be tuned and evolved. Crystal therefore calls for reflection workshops to be held after every delivery so that the methodology is self-adapting.

This relates to the concept of Double Loop Learning, as applied to software development: Single Loop learning describes the plan-do-check-adjust cycle where we learn and increase the efficiency of what we are doing. Double Loop learning is when we step back and question our assumptions and goals and revise them.

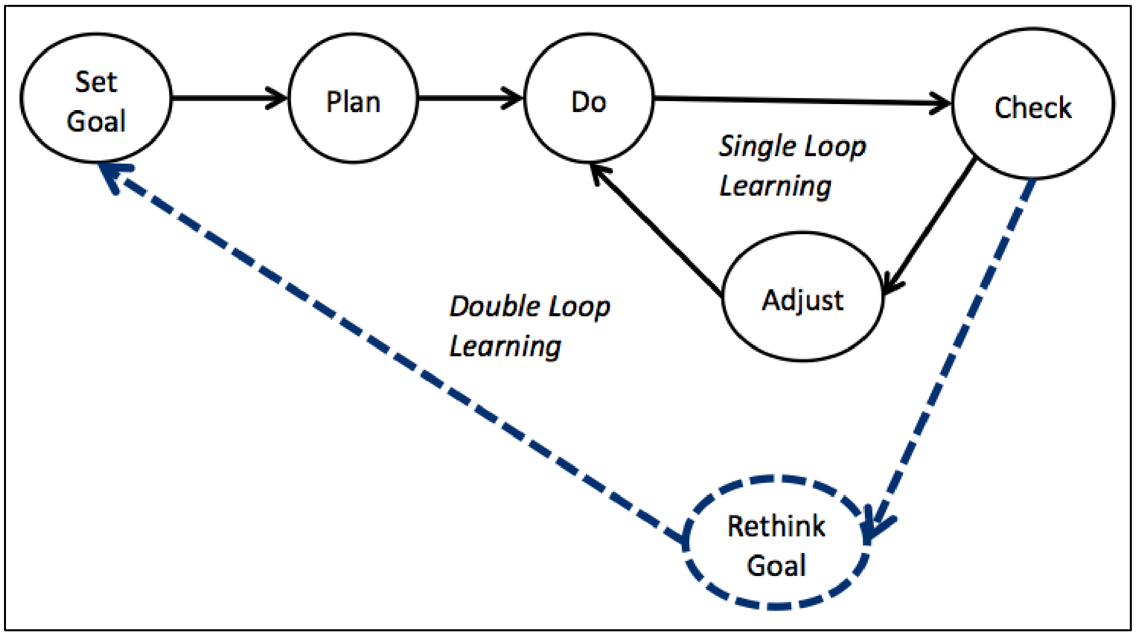


Figure 17 - Simple Representation of Double-Loop Learning

As an example of this, consider code reviews in a software development organization that has a process in place that calls for software inspections. This process may include an onerous series of tasks such as the manual preparation of code review packages. There are so many components, checklists, and forms required to be part of the package that it may take a developer a whole day (or from a project management perspective, a person-day worth of effort) to produce this package. Then perhaps several days elapse before a meeting can be scheduled for all of the necessary players to convene and perform the code review. Finally, action items must be documented, implemented, and verified.

Traditional improvement efforts will focus on automating this process and making it more efficient. The result of single-loop learning may be process enhancements and automations to speed up the code review process for example by automating the review package generation task.

On the other hand, a team that exhibits characteristics of double loop learning will question the goal of the inspection process itself. What is the return on investment (ROI), or value-add, that this inspection process brings to the development effort? It may find that the intent is to simply detect and correct coding defects. The team may react by eliminating this process altogether and adopting the use of pair programming (as a flavor of real-time code inspection) in conjunction with static analysis tools, and even arrange for customer demonstrations and user involvement events, in a push to even further attain the goal of detecting software defects at the implementation level as well as at the level of system users' needs.

Classic development approaches, even ones that employ a single-pass waterfall, can exhibit a "light" flavor of this gene: heavyweight processes often call for Lessons Learned activities at the completion of a software project. The problem is that this usually produces a Lessons Learned document that rarely feeds into the next development cycle and has little improvement effect on subsequent development projects. In the context of Agile, however, the sprints are short enough and retrospectives frequent enough so that process adjustment is near-continuous throughout the life of a project.

Another aspect of many agile methodologies is that teams are often empowered to self-regulate workload. Teams are trusted to self-adjust and gradually learn about how much work they can handle in a given period of time (e.g. sprint). In Scrum, the measure of "Velocity" or "Sprint Velocity" represents how much product backlog effort can handle in one sprint. It is established on a sprint-by-sprint basis as the team learns about the project and about working with each other. Typically, team velocity improves as sprint cycles are completed and experience gained.

3.3.5 Customer Involvement

The Agile Manifesto values "customer collaboration over contract negotiation." A more traditional way of doing things would be to lock-in the system requirements early on in the project. Any subsequent change in direction will require contractual changes for "added scope" or "scope change", and formal CCP (Contract Change Proposal) negotiations. Although this type of project control mechanism helps keep the size of the project in check (and thus helping limit growth in costs and schedule) in the end it may mean that a very long time could be spent up front developing, refining, and validating requirements, but the customer may not get the best software product. The customer may get what they agreed to contractually, but not get the software that they really need.

There are two distinct problems with this "fixed requirements" attitude:
- The "ah-hah" moment: *the first time that you actually use the working product. You immediately think of 20 ways you could have made it better. Unfortunately, these very valuable insights often come at the end.* (Deemer et al. 2009) When requirements are locked-in up front, there is no room for making the product better later in the development cycle.
- It is a well-known fact that undetected requirements defects can wreak havoc on a project if detected late in the schedule. Indeed, that is one of the reasons that the Waterfall method came to be, to lock-down and specify the requirements so that later development could proceed without turbulence. However, as discussed previously, in a changing environment requirements must evolve.

Often the customer (e.g. the DoD) interfaces with the contractor's business operations and project managers. Requirements are generated by business analysts (in some industries also referred to as "system engineers" or "domain experts") and are flowed down to the development team. This means that the people implementing the software are at least two or three degrees of separation away from the customer. To make things worse, the customer is often not the end user of the software product. For example, the real end-user might be a soldier in the field.

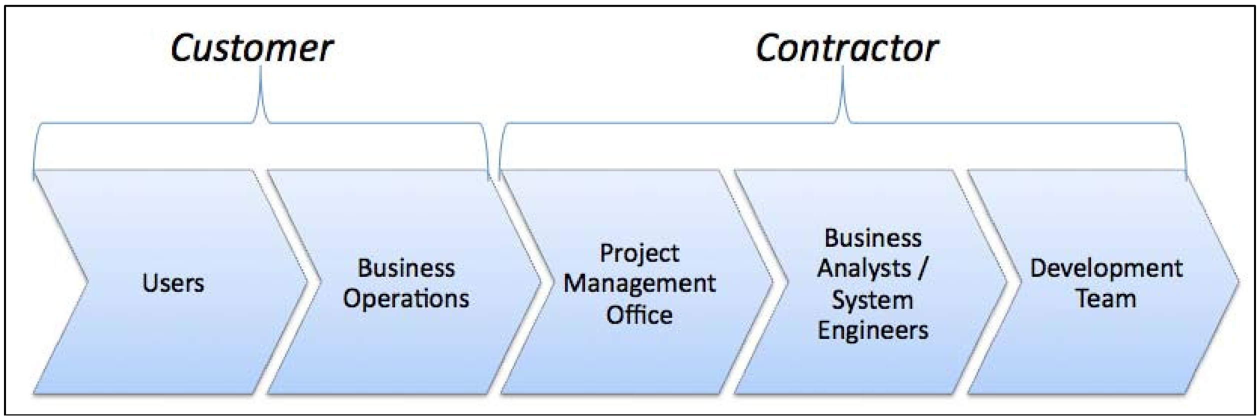

Customer
Contractor
Users
Business Operations
Project Management Office
Business Analysts / System Engineers
Development Team

Figure 18 – Disconnect Between Development and Users

Consider how the waterfall life cycle evolved. First the software development process was divided into phases—planning, analysis, design, coding, testing—and then roles were assigned to each phase—planner, analyst, designer, coder, tester—and then documents were defined for each phase—plan, specification, design document, code, test cases. As roles gave rise to specialized groups and people believed that documentation could convey all the required information to the next group, the barriers to conversation became greater and greater. (Highsmith 2002b)

For this research, the "Customer-focused" gene means accepting changing requirements and including the user and/or customer in the development team (to the degree that this is possible). This can be in daily stand-ups, design reviews and product demos. The customer's availability and input to the development process helps to reduce uncertainty and identify rework early. This requires working with the user/customer to evolve the requirements throughout the process – with later requirements informed by insights gained from earlier development.

An Example of this is the "Product owner" role in the Scrum process, whereby a customer or user proxy is present at all team meetings (the daily Scrum), and is the witness of product demos to give real-time feedback.

3.3.6 Team Dynamics

The "Team Dynamics" gene represents the collection of "soft factors" and effects related to unique agile practices and approaches, and how they affect the development team's performance.

The majority of Agile methods call for frequent meetings to allow teams to self-organize, prioritize, and assign tasks, while communicating roadblocks. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation (Allen & Henn 2006). Practices such as 'pair programming,' and attitudes such as 'collective code ownership' also are claimed to have positive effects on project performance.

A unique team dynamic that emerges in agile teams is a distinctive "social/schedule pressure" effect – As teams convene frequently (on a daily basis, in most cases) to report on what they are doing, what they have done, and what they plan to do next, a certain level of peer pressure comes into play, driving individuals to perform at a higher productivity. Additionally, when developing in short increments/sprints, the team experiences more frequent bouts of schedule pressure coinciding with the end of each iteration, as opposed to a single end-of-project schedule pressure effect.

Another well-known agile practice is that of Pair Programming. In pair programming, two programmers jointly produce work products (e.g. code, design documentation). The two programmers periodically switch roles as "the driver" controls the pencil, mouse, or keyboard and writes the code. The other partner continuously and actively observes the driver's work, watching for defects, thinking of alternatives, looking up resources, and considering strategic implications. The partners deliberately switch roles periodically. This practice has been reported in most cases to reduce productivity but produce higher quality code.

Team co-location and open workspaces and environments are also preferred in Agile teams. These also help promote the flow of information between and drive team performance through transparency.

3.3.7 Continuous Integration

Agile methodologies often include certain policies and approaches to Configuration Management (CM,) as well as a high level of automation of as many aspects of the development process as possible, to speed up the development by eliminating repetitive and manual tasks.

Traditionally, the popular CM approach was to have different teams develop portions of the software in isolated environments, and integrate their work products later, towards the end of the development cycle. In the mid-1990s, Microsoft moved beyond the Waterfall approach to a "synch-and-stabilize" process which relies primarily on daily builds for frequent synchronizations and incremental milestones for periodic stabilizations of the products under development (Cusumano & Smith 1995). This is one of the early examples of automating nightly software builds and frequently integrating various pieces of software under development to detect early conflicts. That is one example of Continuous Integration.

This gene also includes test automation. Testing can be automated at various levels, from unit testing, to the of system-level tests. Test automation means having software perform the testing that would otherwise need to be done manually. Once tests have been automated, they can be run quickly and repeatedly. This is the most cost effective method for software products that have a long maintenance life, which often requires copious regression testing to verify that there is no breakage in baseline functionality introduced by added features.

Traditionally software houses employ a different set of standards and practices when it comes to software development versus software delivery. In fact, it often is the responsibility of two completely different teams, whom often use a different set of tools and environments. The delivery team's focus is to compile and ship working software, not necessarily developing code. However Continuous Integration calls for a shared environment for both - integrating and automating as much as possible. Some principles of Continuous Integration are:
Development:

1) Stay current (merge code early and often)
2) Deliver working code (don't submit changes that break the build)
3) If the code changes, write a new test

Delivery:

1) Build from a clean environment.
2) Fixing the build is top priority.
3) Tests should be thorough and repeatable, and run automatically to verify the build.

3.4 Summary

Research into agile methodologies has revealed a set of recurring patterns and similarities across the software industry. We have captured the essence of what we think makes a software project Agile in the following seven characteristics we now dub the "Genome of Agile" – see Table 4 - The Genome of Agile.

Gene Name	Short Description
Story/Feature Driven	Break up of project into manageable pieces of functionality; sometimes named "features", "stories", "use cases", or "threads".
Iterative-Incremental	Development is performed in repeated cycles (iterative) and in portions at a time (incremental.)
Refactoring	Refinement of the software design and architecture to improve software maintainability and flexibility.
Micro-Optimizing	Teams are empowered to modify aspects of the process or dynamically adapt to changing circumstances. Small improvements and variable changes are made frequently and as needed.
Customer-Involvement	Customer/User involved in demonstrations of functionality to verify/validate features. Higher frequency feedback and clarification of uncertainty. Availability to participate in development meetings.
Team Dynamics	"Soft" factors related to the project team. Daily meetings, agile workspaces, pair programming, schedule/peer pressure, experience gain, etc.
Continuous Integration	Policies and practices related to Configuration Management, and build and test automation.

Table 4 - The Genome of Agile

In the following chapters, we will introduce the System Dynamics methodology and put it to use in modeling the software project-system, including the seven agile genes.

4. System Dynamics and its Applicability to Software Project Management

Feedback processes govern all growth, fluctuation, and decay. They are the fundamental basis for all change. They allow new insights into the nature of managerial and economic systems that have escaped past descriptive and statistical analysis.

- *Jay Forrester, creator of System Dynamics*

This chapter provides a high-level introduction to System Dynamics. It follows by taking a look at two project-related phenomena, the "rework cycle" and "Brooks' law", that have been studied in prior research with system dynamics. We conclude this chapter by presenting examples of how System Dynamics has been used in industry to study software project behavior.

4.1 Introduction to System Dynamics

Dr. Jay W. Forrester at the Massachusetts Institute of Technology created System Dynamics in the 1960s. It is a method for modeling and understanding the dynamic behavior of complex systems. Originally applied to the study of management and engineering systems, this powerful approach has found its way into many other fields, which study complex systems such as social, urban, economic, and ecological systems, amongst others.

First, let us define the word **system**. My favorite definition comes from Russel Ackoff, a professor at the Wharton School, and a pioneer in the field of Systems Thinking. In "Towards a System of Systems Concepts" (Ackoff 1971), Ackoff presents perhaps the best attempt at defining the concept of *system*. It can be summed up in the following definition:

A system is a set of two or more elements, and satisfying the following conditions:

- Each element can affect the behavior of the whole.
- The way each element affects the whole depends on at least one other element – i.e. <u>no element has an independent effect on the whole</u> (the parts are interconnected.)
- If you take any groups/subsets of elements and arrange them in any fashion, form subgroups in any way at all, then the subgroups will have these two properties:
    - Every subgroup can affect the behavior of the whole
    - The way each subgroup affects the whole depends on at least other subgroup (no subgroup can have an independent effect)

An oversimplified version of this would be: "A system is a whole which cannot be divided into independent parts."

If we study this definition carefully, it implies two things about systems that are extremely important for management:

1) The essential properties of any system are properties of the whole which none of its parts have. Therefore when a system is taken apart it loses its essential properties. E.g. if we disassemble a car, we no longer have a car but a collection of car parts.
2) No system is the sum of the behavior of its parts; it is a product of their interactions. When a system is disassembled, not only does it lose its properties, but so do all of the parts. E.g. an engine moves a car, but alone it moves nothing.

Therefore, we cannot study the development process without considering the development team (one criticism of CMM is that it looks at individual programmers as replaceable parts.) We cannot study the development team without understanding the software they are developing. And so on…

System Dynamics allows us to model the structure of a system and all its “parts”, including the time-delayed relationships among its components. The underlying relationships and connections between the components of a system are called the **structure** of the system. The term **dynamics** refers to change over time. If something is dynamic, it is constantly changing. A dynamic system is therefore a system in which the variables interact to stimulate changes over time. The way in which the elements or variables composing a system vary over time is referred to as the **behavior** of the system. (Martin 1997)

Central to System Dynamics is the concept of **feedback loops**, and the concepts of **stock** and **flow** (also referred to as **level** and **rate**). A stock is something that accumulates (or drains) over time. A rate describes the quantity per unit-time at which a stock accumulates (or drains). Feedback loops are the cause-and-effect chains in the system through which a change in one variable creates a circular feedback, ultimately affecting itself.

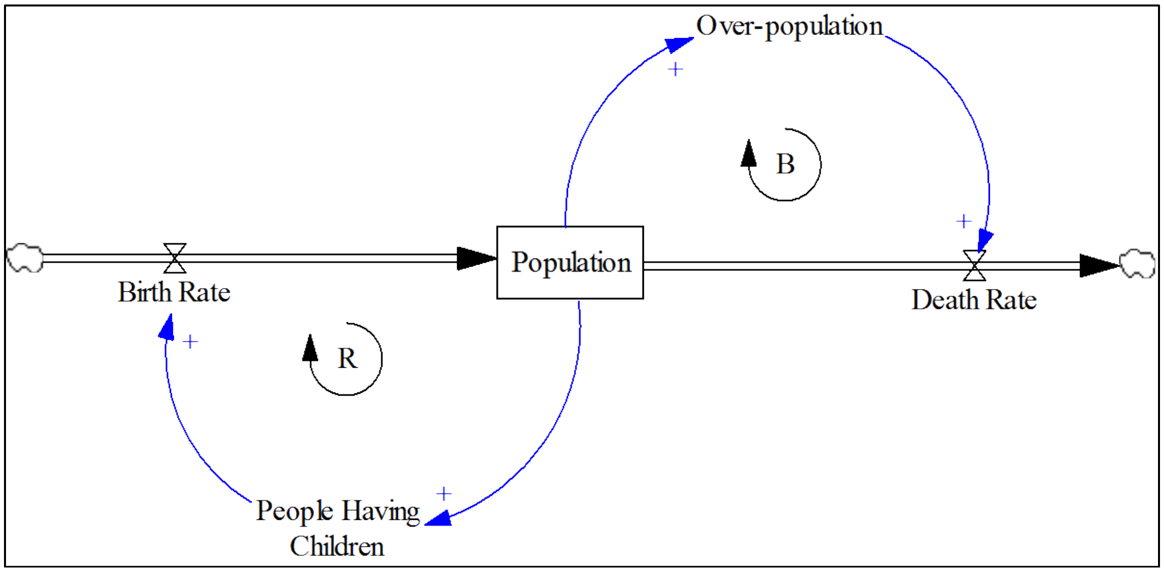


Figure 19 - Simple System Dynamics Example

To demonstrate these concepts, Figure 19 presents a very simple example of a single-stock system:

- *Population*: is the stock that represents a human population.
- *Birth Rate* and *Death Rate* are the rates at which the stock accumulates and drains, respectively.
- The loop annotated with "R" is a *positive feedback loop*, also called a *reinforcing feedback loop* because it drives growth in the system. In our example: the more *Population*, the more *People Having Children*, and thus a higher *Birth Rate*, leading to even more *Population.* In other words, an increase in population triggers more increase in population.
- The loop annotated with "B" is a *negative feedback loop*, also called a *balancing feedback loop* because it has a stabilizing effect on the system. In our example: the more *Population*, the greater *Over-population*, leading to a higher *Death Rate*, which causes a reduction in *Population.* In other words, an increase in population triggers a decrease in population.

The previous example is a very simple single-stock system with only two feedback loops. The study of a software development project on the other hand includes many stocks, variables, and loops. The type of large scale software projects we are interested in belong to the class of complex dynamic systems that, according to John Sterman (Sterman 1992), exhibit the following characteristics:

- They are complex and consist of multiple components.
- They are highly dynamic.
- They involve multiple feedback processes.
- They involve non-linear relationships.
- They involve both "hard" and "soft" data.

In our case, hard data would be measurable data such as cost, and soft data would be more intangible data such as 'employee motivation'.

4.2 The Rework Cycle

The conventional view of a project is as a collection of predefined tasks. Based on a predetermined work rate (number of tasks that can be accomplished by the project team per unit of time,) the project manager can project an estimate of how long it takes to complete development. Figure 20 shows the SD representation of a project model based on these assumptions, using stocks and rates as follows:

- *Work To Do*: The amount of work to be performed.
- *Work Done*: The amount of work completed.
- *Productivity*: The amount of work that can be done per-developer, per unit of time.
- *Number of Developers*: The number of developers available.
- *Work Rate:* The rate at which *Work To Do* is drained, and at which *Work Done* accumulates. It is the product of *Productivity* by *Number of Developers.*

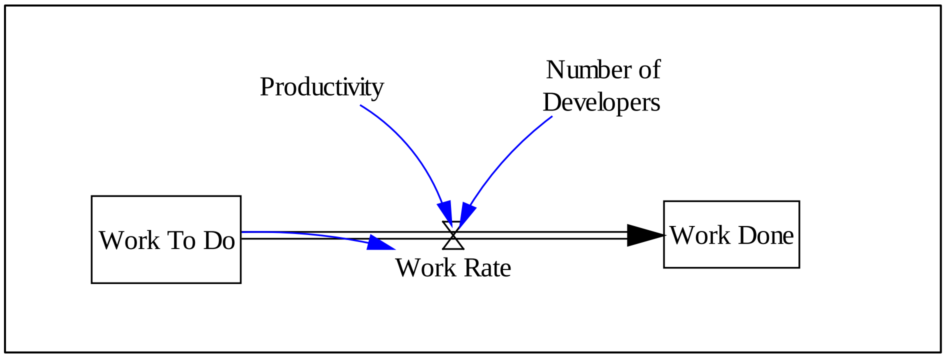


Figure 20 - Traditional View of Development Work Completion

If we execute this model with the following initial values:

- *Work To Do* = 8000 Source Lines of Code (SLOC)
- *Productivity* = 160 SLOC per Month, per person
- *Number of Developers* = 10 people

Then our *Work Rate* will be a steady 1600 SLOC per month, and the project will take five months to complete (the time at which the level of *Work To Do* reaches zero), as shown in the Figure 21 simulation results from executing this model.

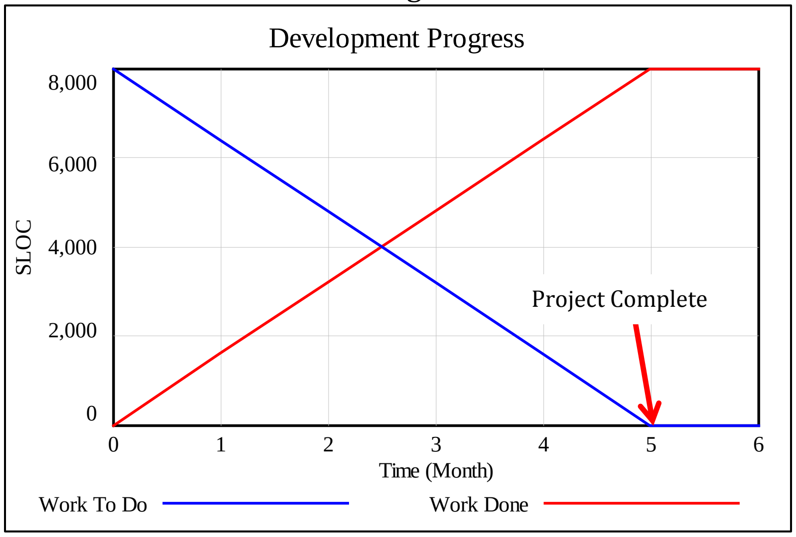


Figure 21 - Development Progress Based on Simple Project Model

However, it is clear that this model is overly simplistic. It does not take into account the fact that *Work To Do* is constantly changing, not only decreasing because of work being completed, but also fluctuating because defects, errors, and other forms of "rework" are generated during development.

Evolving this model to add in the concept of rework yields the following version of the model in Figure 22, where a *Fraction Correct and Complete* (FCC) dictates the percentage of completed work that is correct and defect-free, ending up in the stock of *Work Done Correctly*. The remainder of the work is either incorrect or incomplete and requires rework, and thus makes it into the *Undiscovered Rework* stock.

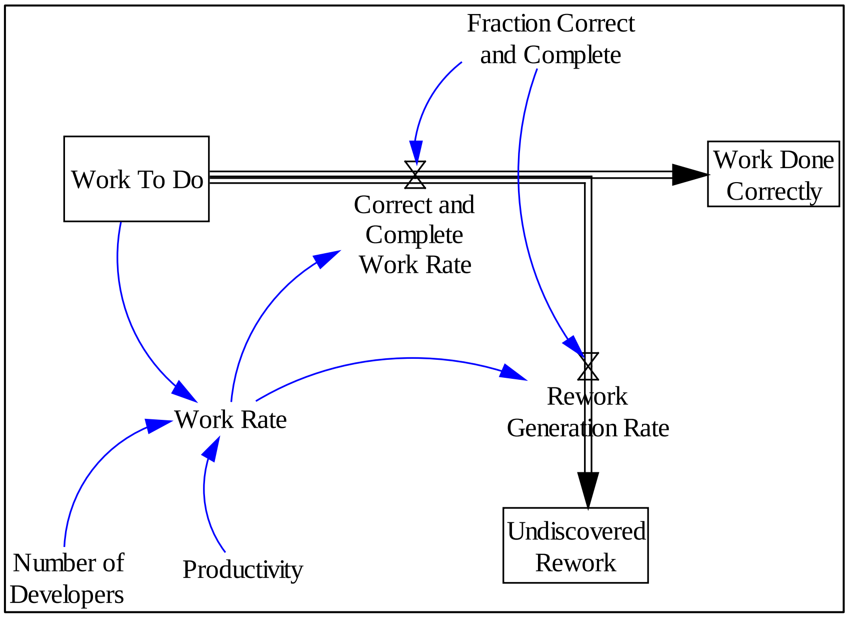


Figure 22 – Basic Project Model with Rework Generation.

Continuing with our example, assuming an FCC of 80%, we find that 1600 additional SLOC worth of rework have been introduced into the system by the fifth month, as shown in the graph in Figure 23, generated by executing this revision of the model.

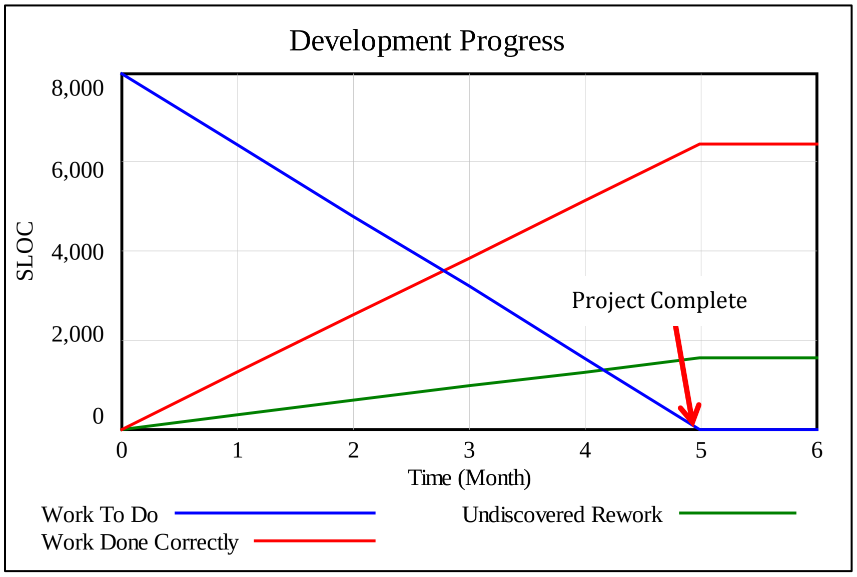


Figure 23 -Development Progress and Undiscovered Rework

Note that the project is considered complete when *Work To* Do reaches zero at month five. However we also observe that at month five, there are yet 1600 SLOC in *Undiscovered Rework.* In other words, 20% of the completed work needs to be re-worked, but the project team is not yet aware of this. It is also important to note that while rework remains undiscovered, project management can overestimate progress, thinking that by month 5 the project will be complete.

In real projects however, rework is discovered at various stages through validation and verification – Let us further refine this model to introduce an element of rework discovery. As code is inspected, tested, integrated, deployed, etc. rework is discovered and added to the collection of *Work To Do*. Figure 24 shows the model revised to add rework discovery, assuming a nominal *Time to Discover Rework* of one month.

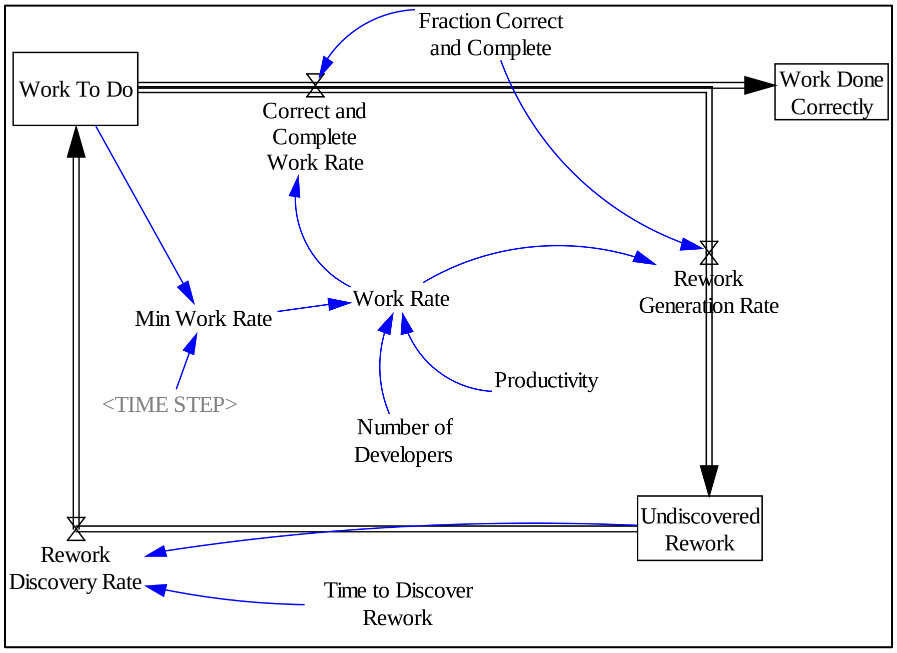


Figure 24 - Basic Project Model with Rework Generation and Discovery

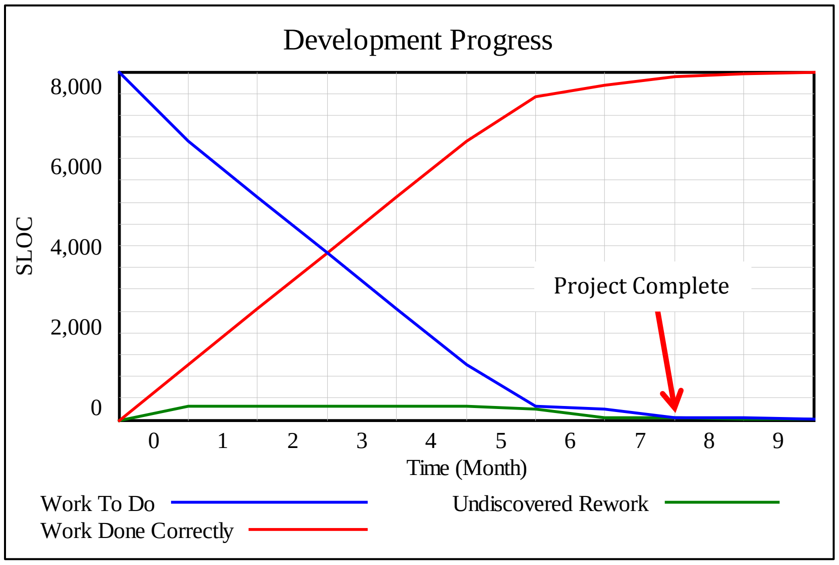


Figure 25 - Development Progress with Rework Generation and Discovery

This completes a simplified model of the classic "Rework Cycle". Due to the rework effects, a project that was planned to complete in five months now takes up to ten months, as shown in the output graph in Figure 25. This in part explains the "90%" phenomena: often, as a project nears its end, it seems to be stuck at "90% complete" for a long time, without apparent progress. The "90% syndrome" is due to the effect of rework discovery: especially at the tail-end of a Waterfall project where system level testing and integration activities ramp up and begin to identify defects and rework. In our example, our project is stuck at 90% from months 6 to 10. As software components are completed and integrated, interface problems are identified, as well as user/customer related issues when the final product is first presented to them.

Additionally, as rework and defects remain undiscovered in the system, new development work built on top of these may end up needing rework as well. We call this the "Errors upon Errors" effect: A second order feedback describing the compounding nature of undiscovered rework, as shown in Figure 26. The more undiscovered rework in the project, the more likely that new work built on top of it will require rework as well. In our example, when adding this loop even more rework is generated, and now the project takes about 16 months to complete (see Figure 27)!

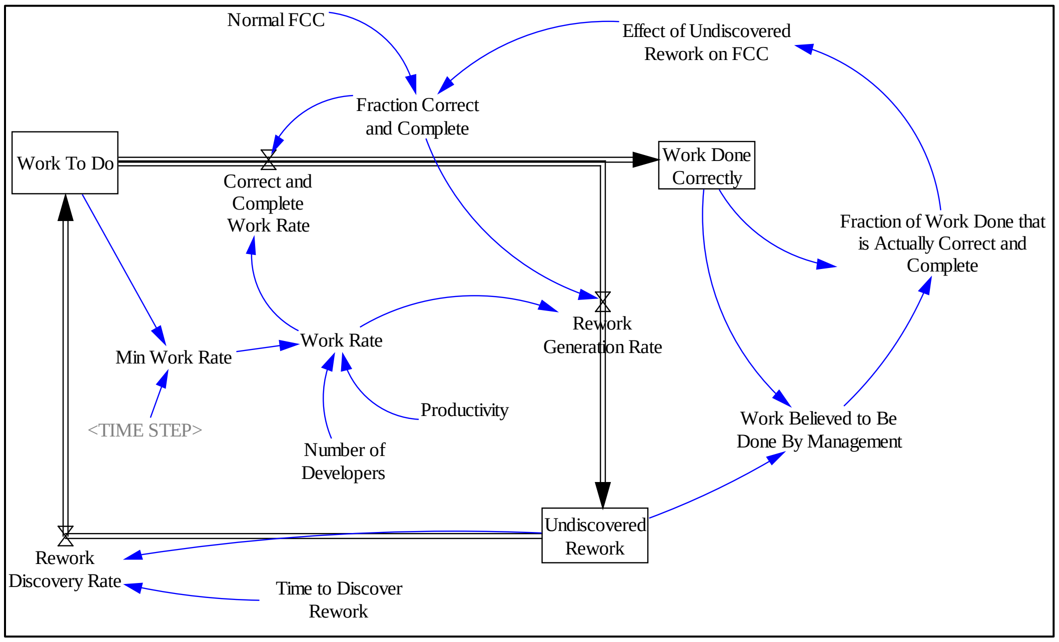

Normal FCC
Effect of Undiscovered Rework on FCC
Fraction Correct and Complete
Work To Do
Work Done Correctly
Correct and Complete Work Rate
Fraction of Work Done that is Actually Correct and Complete
Rework Generation Rate
Work Rate
Min Work Rate
<TIME STEP>
Productivity
Number of Developers
Work Believed to Be Done By Management
Undiscovered Rework
Rework Discovery Rate
Time to Discover Rework

Figure 26 - Rework Cycle Model with Errors-Upon-Errors Effect

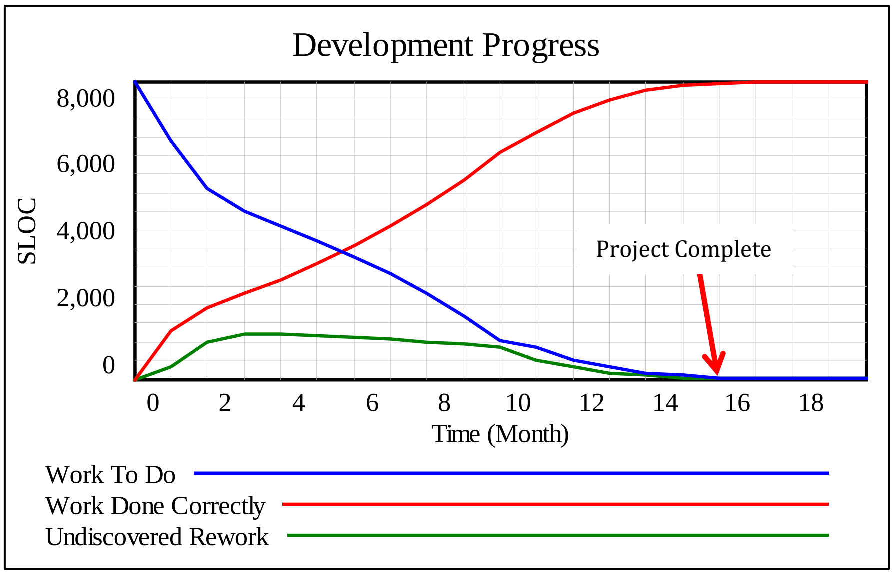


Figure 27 - Development Progress with Errors Upon Errors Effect.

In the field of Strategic Project Management, the rework cycle is seen as being at the root of the dynamics behind project performance. Errors and undiscovered rework lead to more work in the form of further cycles (iterations) which are unplanned. Traditionally, project managers deal with this situation by adding a "buffer" to the project in the planning phase, either in the form of a schedule buffer (such as practiced in Critical Chain Path Management) or a cost buffer (e.g. the 'management reserve' in EVM) to fund extra staffing or overtime effort.

The clear messages from the rework cycle are (1) improve fraction correct and complete (do it right the first time); (2) discover rework as soon as possible (avoid the "errors upon errors" effect); and (3) incorporate estimates of undiscovered rework in project status. (de Weck & J. Lyneis 2011) Fine-tuned System Dynamics models can be used to estimate the potential impact of rework in projects and plan accordingly.

4.3 Brooks' Law

In one of the classic works of software project management, "The Mythical Man Month: Essays on Software Engineering", Fred Brooks first articulated what has now come to be known as "Brooks' Law":

Adding manpower to a late software project makes it later. (Brooks 1975)

Brooks attributes this phenomenon to two main factors. One is the fact that new people on the project take time to become productive, as there is a learning curve associated with introducing new team members. Regardless of a developer's experience and depth of technical expertise, there is usually project-specific knowledge that is unlikely to be known to a new person. During this "ramp up" time the team is less productive as a whole. This is due not only to the initial low productivity of new team members, but is also due to the time that experienced team members must spend mentoring and coaching new staff, causing the experienced staff to be less productive themselves. The second factor has to do with the communication and coordination costs that increase as the team size grows.

In the context of our SD example model from the previous section, we can describe this effect by decomposing our *Number of Developers* into two stocks of *Inexperienced Developers* and *Experienced Developers*. Over time, *Inexperienced Developers* gain experience at a rate dictated by *Time to Gain Experience*. Also, we add a *Hiring Rate* and a *Staff Departure Rate* to incorporate the effect of people leaving and joining the project. This simple staffing model is depicted below in Figure 28.

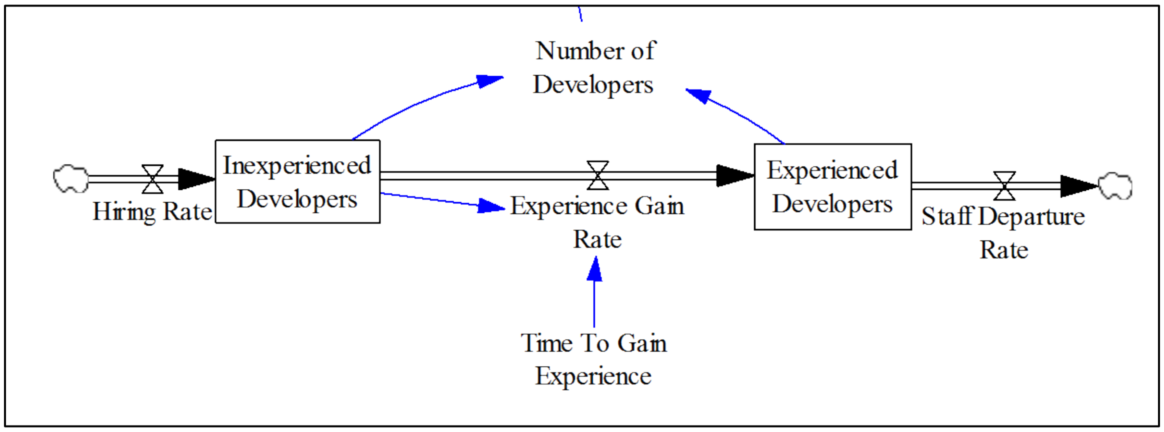


Figure 28 - Simple Project Staffing Model

Now, the other problem with simple project management models is that they employ non-dynamic values for *Staff*, and do not incorporate the effects of a dynamically changing staff composition on Productivity and FCC:

- Staff (Number of Developers) often changes, especially on long-term programs. Moreover, not all developers are equal. Experience mix and learning curves play a part in how much "effective staff" are actually applied to the work.
- Productivity is affected by the experience mix of the team.
- The same is true for FCC, as a higher ratio of experienced to inexperience staff will lower defect generation.

Simple project models rely on a static, average, value of productivity, such as the 160 SLOC per person-month used in our example in section 4.2. Usually this is a historical measure of an organization's performance on other projects. Let us improve this by extending the previous model to incorporate the effects of staff on project performance. A straightforward approach to this is modeled in Figure 29.

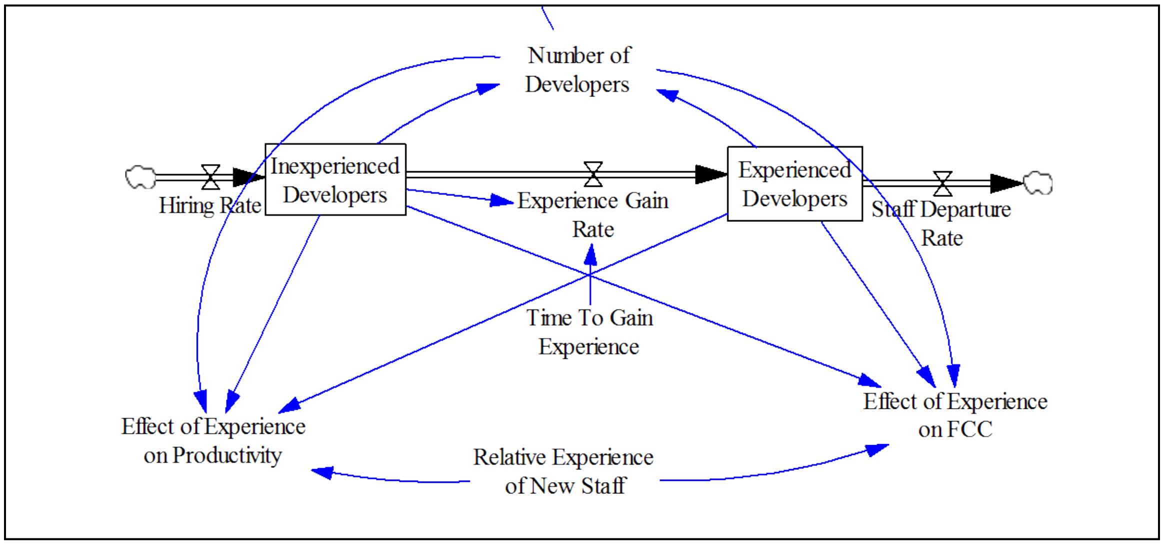


Figure 29 - Experience Mix Added to Model

Relative Experience of New Staff represents the answer to "how experienced are new staff members compared to existing project staff members?" This can vary greatly from project to project, and organization to organization. For example a new developer on a Java project in the Air Traffic Control (ATC) domain may be a Java coding guru with years of experience, but be a new hire to the company and have no experience with ATC, the company culture, etc. For our modeling purposes, *Relative Experience of New Staff* is modeled as an exogenous variable.

Using this relative experience ratio we can formulate values for the *Effect of Experience on Productivity* and the *Effect of Experience on FCC*. The effect is based on the fraction of staff which are inexperienced and the *Relative Experience of New Staff*. Adding these effects in our model now shows how staffing dynamics affect controlling parameters of the rework cycle (Productivity and FCC.) Choosing a value of 20% for *Relative Experience of New Staff*, and ten new developers at the outset of the project, and zero experienced developers, we see the following model behavior.

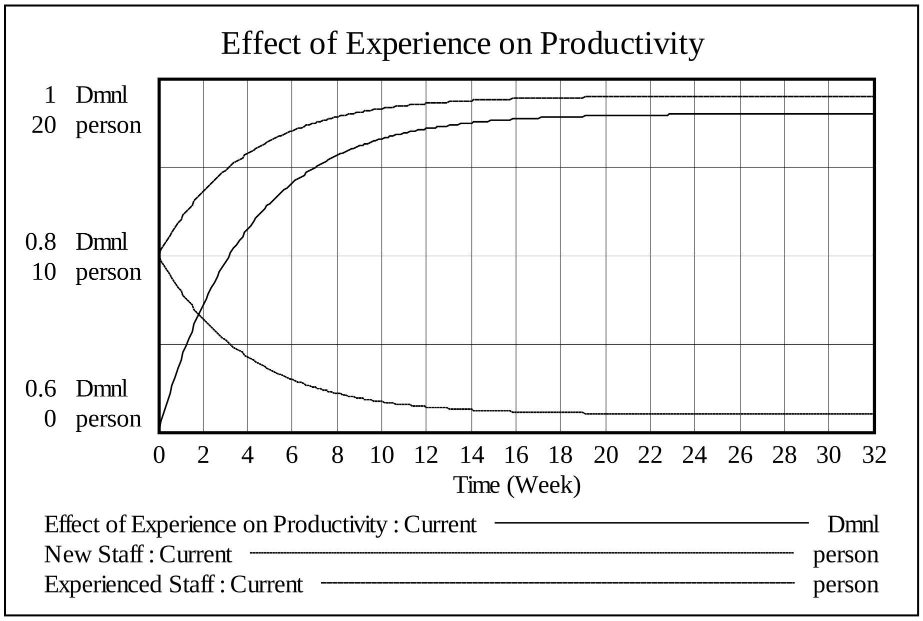


Figure 30 - Effect of Experience on Productivity

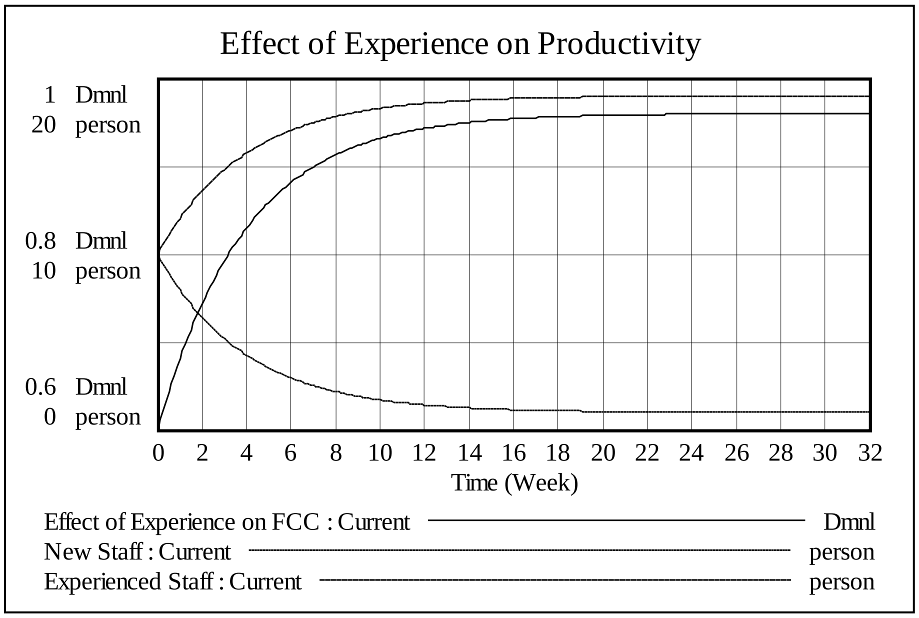


Figure 31 - Effect of Experience on Fraction Correct and Complete

Figure 30 and Figure 31 show how Productivity and FCC suffer due to staff inexperience, but as developers gain experience and assimilate into the pool of experienced staff, the average productivity sees a marked improvement. These graphs plot the *Effect of Experience on FCC*, the number of *New Staff*, and the number of *Experienced Staff* to provide a visual indicator of improvement in FCC as the team composition evolves towards mostly experienced staff. Note that the *Effect of Experience on FCC* is a fraction, and modeled as a dimensionless unit (denoted “Dmnl” in these graphs). For example, an *Effect of Experience on FCC* of 1.05 means that this has a 5% improvement effect of FCC.

Our simple case so far assumes no staffing changes during the span of the project. However, in real projects (especially large-scale software engineering), staff churn is a reality that project management must fully consider as an integral part of the project-system. Let us see what effect staff experience has on project completion time (Figure 32): It now takes about 22 months to complete all of *Work To Do*, for a project that "ideally" took five months to do back in Figure 23!

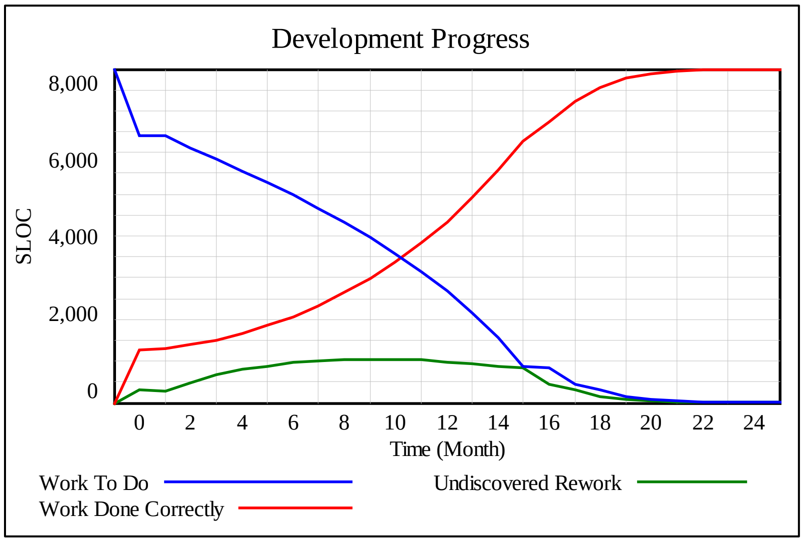


Figure 32 - Project Completion With Staff Experience

Now, let us see what happens when the management, at month number 16, decides to "rescue" the project by adding 5 more developers to the team. The results are shown in Figure 33. It now takes up to 24 month to complete.

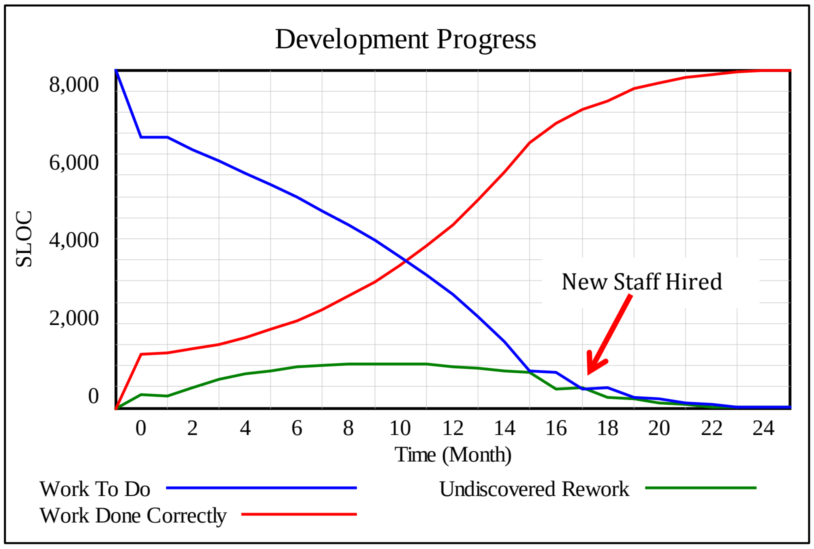


Figure 33 - Project Performance with Late Hiring (Brooks' Law)

Although a contrived example, this model illustrates the behavior behind Brooks' law. The lesson for software project managers here is again to look for ways to improve FCC and Productivity – Adding staff late in the project simply makes things worse because it lowers productivity and increases defect generation, in aggregate. We can observe the effect of management's hiring decision for example by comparing the graphs for *Rework Generation Rate* for both cases (with and without hiring at month 15). The comparison is shown in Figure 34. This extra rework, generated due to the addition of new staff, increases the amount of work that needs to be done for the project to complete.

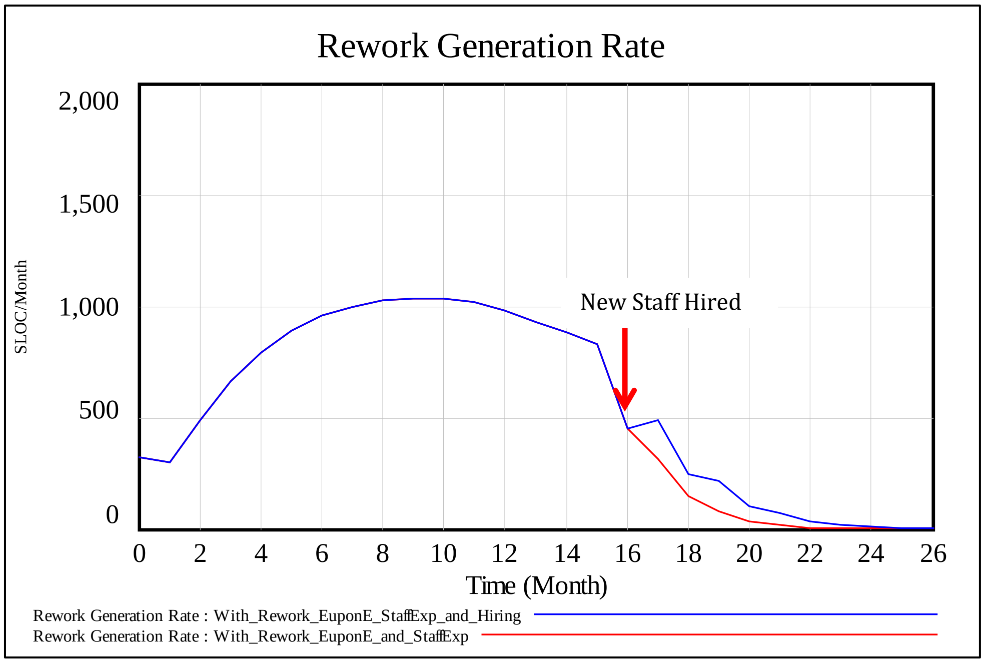


Figure 34 - Rework Generation, With Increased Rework Generation Due to New Staff

4.4 Strategic Project Management with System Dynamics

System Dynamics was famously used by Pugh-Roberts/PA Consulting to diagnose the causes of cost and schedule overruns on an Ingalls Shipbuilding (a division of Litton Industries, Inc.) multibillion-dollar shipbuilding program in the 1970s, leading to a $447 million dollar settlement for the shipbuilder (Cooper 1980). Since then, they have applied system dynamics in dozens of contract disputes related to cost and schedule overruns on very large complex engineering projects.

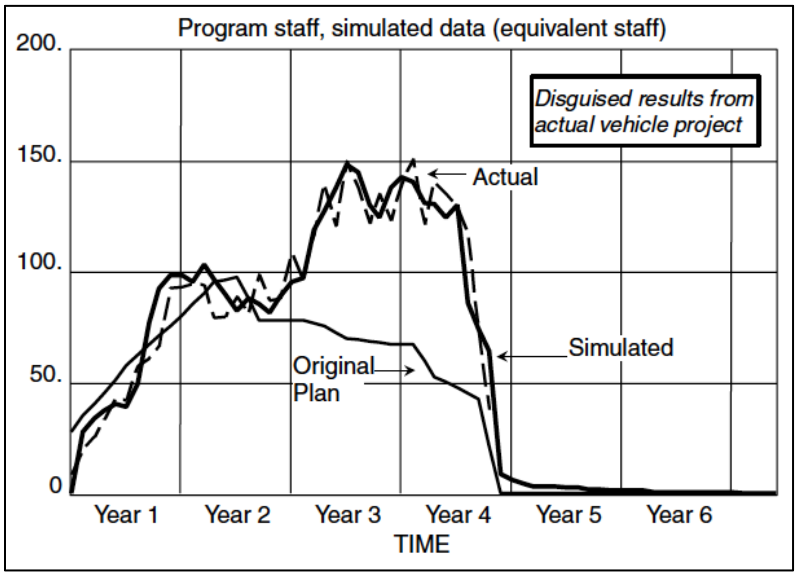


Figure 35 - SD Model Reproducing Behavior of Real (Disguised) Project

Figure 35 shows the project staffing profile on a real multi-million dollar aerospace and defense project that ran into such trouble (J. M. Lyneis et al. 2001). Note that the model was able to accurately reproduce the system's behavior, which was well off plan. Furthermore, System Dynamics goes beyond post-mortem analysis of troubled projects. System Dynamics models can be used to aid in a proactive, strategic/tactical management of design and development projects.

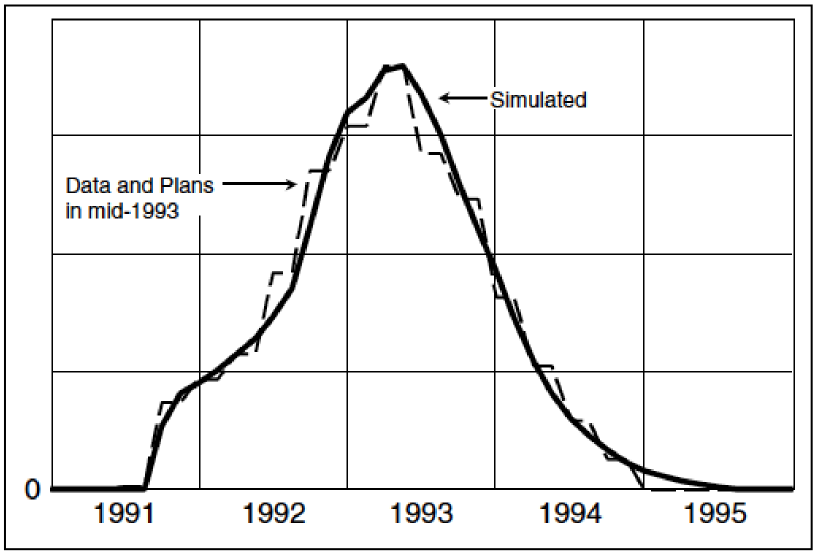


Figure 36 - Planned vs Simulated Cumulative Effort for Peace Shield Air Defense System Project

This is illustrated by a case study of the Peace Shield Air Defense System, a $1 billion-plus Hughes Aircraft (now part of Raytheon) program. On this project, a model (at the heart of which was the rework cycle) was used to support the project bid, to identify and manage risks, and to assess the benefit of several process and organization changes, which were implemented on the project. Upon completion, the actual project results mirrored the project plan, on time and within budget (Figure 36). The Acting Assistant Secretary of the Air Force for Acquisition, was quoted as saying ''In my 26 years in acquisitions, this is the most successful program I've ever been involved with, and the leadership of the U.S. Air Force agrees.'' (J. M. Lyneis et al. 2001)

4.5 Summary

This chapter introduced the Systems Dynamics approach to modeling socio-technical systems. After a brief overview of stocks, flows, and feedback loops – the basic elements of system dynamics models - we took a look at the rework cycle, a model construct that has been shown to be at the root of project performance in various System Dynamics research efforts. We've also illustrated System Dynamics' applicability to the domain of software project management by using the rework cycle along with a simple staffing model to demonstrate Brooks' law (*adding manpower to a late software project makes it later*). We concluded by presenting real-world examples of the use of System Dynamics for strategic project management. With this in mind, we next proceed to model the "software development system", while incorporating the structure and feedbacks that capture the seven "Agile Genes" presented in section 3.3. This model can be part of an "ongoing learning system" that will aid in the design and deployment of software development processes for projects that seek to enjoy the benefits of Agile Software Development.

5. Modeling the Dynamics of Agile Software Projects

This chapter presents the Agile Project Dynamics (APD) model. Following the research into Agile methodologies and our formulation of the agile genome in the previous chapters, we now employ System Dynamics to build a model of the software project-system. Figure 37 is a notional illustration of the APD model as a "black box" system. As exogenous inputs to this model are project-specific parameters (system features, staff, number of releases, etc), project-team specific parameters (productivity, rework discovery rate, etc.) and "agile levers" (i.e. policies regarding which agile practices, or 'genes' to be used in the project). Based on these inputs, model simulations capture project performance as output in terms of cost, schedule, and quality.

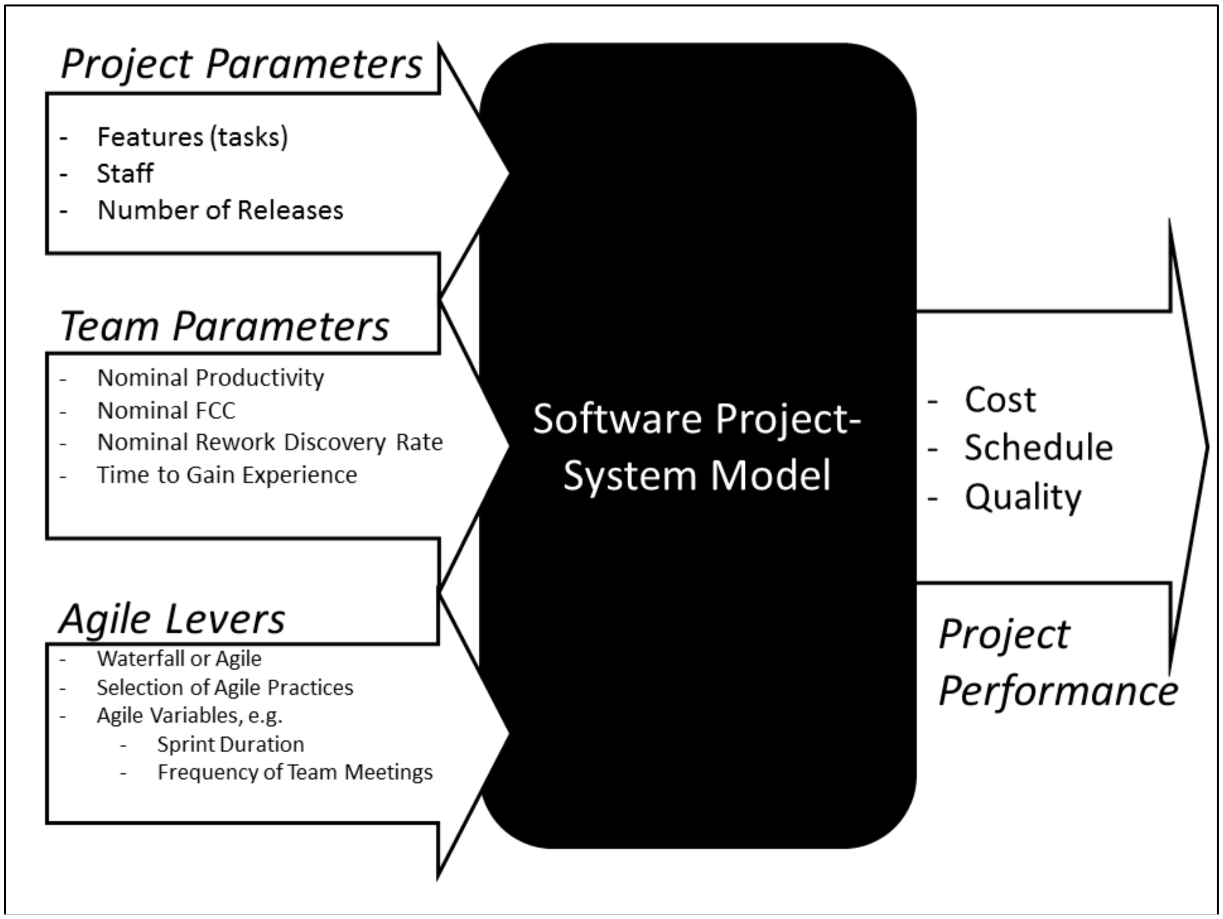
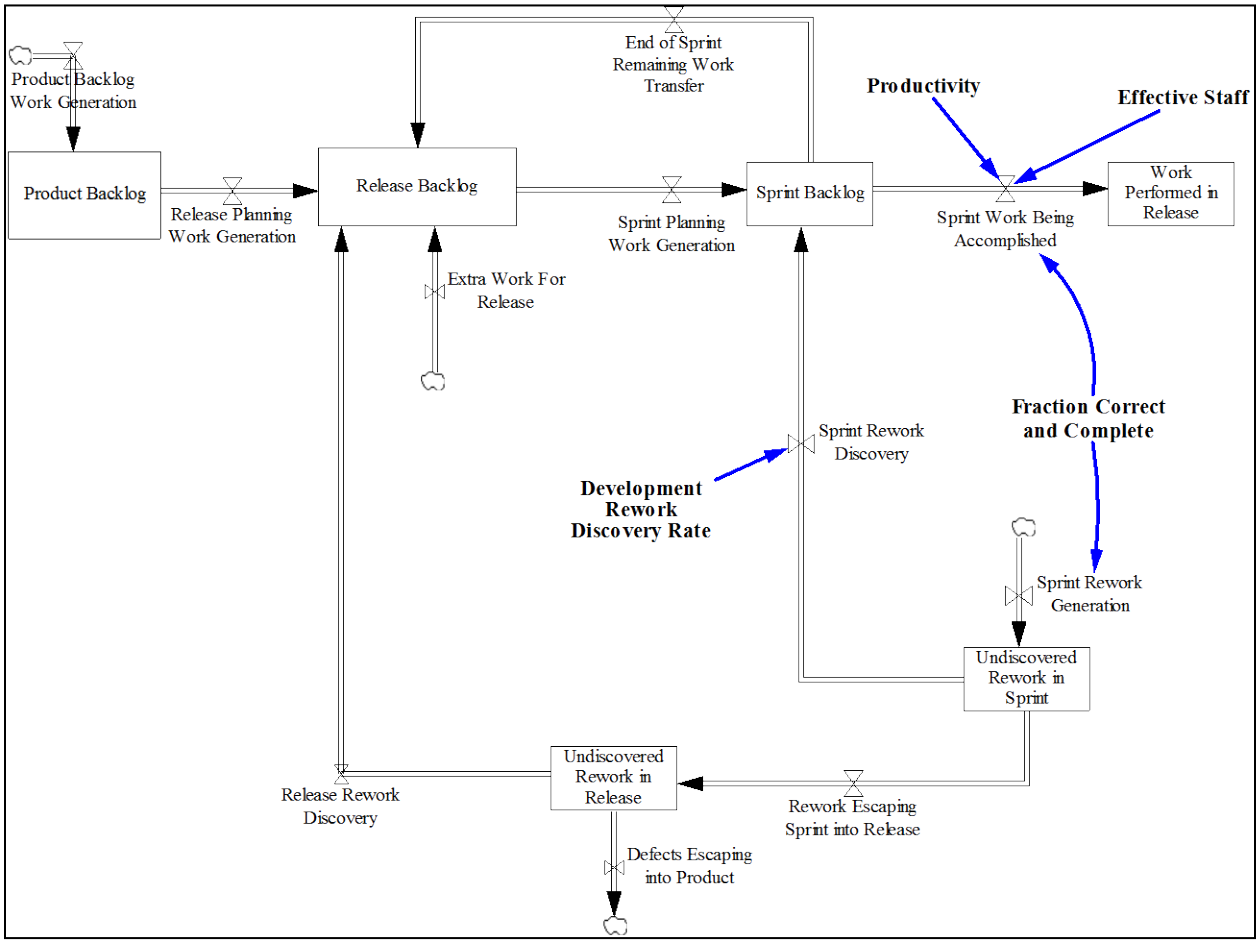


Figure 37 - APD Model "Black Box"

5.1 APD Model High-Level Overview

The APD model is a complex model developed using the Vensim PLE system dynamics modeling tool. It is built using several "views" which allow us to build different subcomponents of the model in isolation, and to link them via the use of "shadow variables". At the core of the model is the "Agile Rework Cycle". This consists of the rework cycle structure extended to support an iterative-incremental development style, or a single-pass waterfall approach – so that we can compare these differences. Figure 38 shows this cycle. Note that in this figure, many variables and causal links have been removed or renamed for clarity and display purposes. This full model structure, complete with all contributing variables and loops, is presented in Appendix 1 – The Full Agile Project Dynamics Model. Sections 5.2.1 and 5.2.2 will later explain how this structure supports the "Feature Driven" and "Iterative-Incremental" gene behaviors.



End of Sprint Remaining Work Transfer
Product Backlog Work Generation
Productivity
Effective Staff
Product Backlog
Release Backlog
Sprint Backlog
Work Performed in Release
Release Planning Work Generation
Sprint Planning Work Generation
Sprint Work Being Accomplished
Extra Work For Release
Fraction Correct and Complete
Sprint Rework Discovery
Development Rework Discovery Rate
Sprint Rework Generation
Undiscovered Rework in Sprint
Undiscovered Rework in Release
Release Rework Discovery
Rework Escaping Sprint into Release
Defects Escaping into Product

Figure 38 - APD Model, Agile Rework Cycle

We characterize our complete model as "complex" because it includes 179 variables (parameters, lookup tables, equations) and a high number of feedback loops. The table below summarizes the number of loops that some of our key parameters are involved in:

Variable	Number of Loops
Fraction Correct and Complete	5164
Sprint Work Being Accomplished	977
Sprint Backlog	619
Product Backlog	611
Sprint Rework Discovery	242
Release Backlog	212
Undiscovered Rework In Sprint	111
Sprint Rework Generation	90
Productivity	68
Development Rework Discovery Rate	65
Effective Staff	12

Table 5 - Loop Counts for a Sample of Variables in the APD Model

Since there are so many loops driving the behavior of the system, we cannot hope to explain them all in this document, however what follows is a description of how all of the pieces 'fit together' in this model.

Each of the seven genes' has its own set of dynamic effects on these controlling variables of the "Agile Rework Cycle" (Figure 38). In other words, each agile practice or characteristic has both positive and negative effects on:

- the rate at which work is accomplished,
- the rate at which defects are produced,
- the rate at which defects are discovered, and
- the rate at which working software is released.

Each gene, when possible, is modeled in its own view, and connected back to the agile rework cycle using shadow variables. Shadow variables are a mechanism in the Vensim modeling tool whereby a variable that appears in one model view can be referenced (or "imported") into another model view. This allows linking model structure across views, and building 'sub-systems' of the model in multiple views.

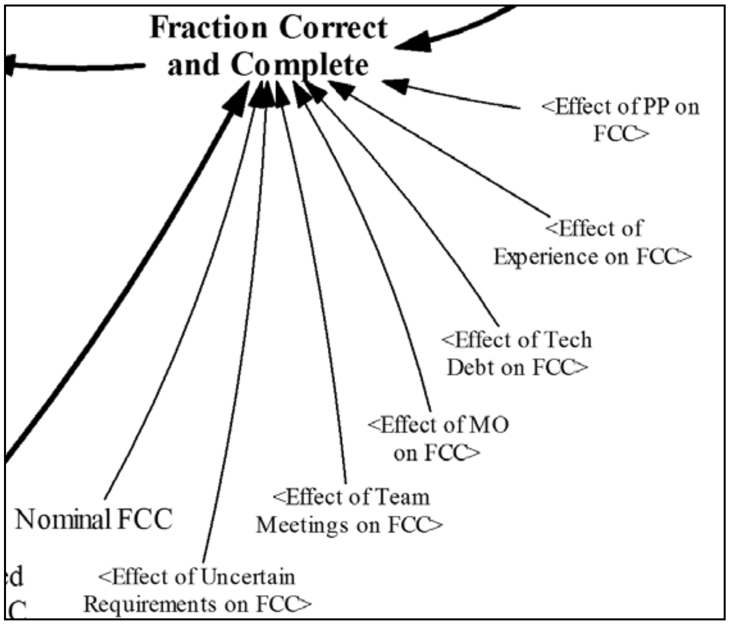

Fraction Correct and Complete
<Effect of PP on FCC>
<Effect of Experience on FCC>
<Effect of Tech Debt on FCC>
<Effect of MO on FCC>
<Effect of Team Meetings on FCC>
Nominal FCC
<Effect of Uncertain Requirements on FCC>

Figure 39 - Shadow Variables Linked to FCC in APD Model

For example, Figure 39 shows how Fraction Correct and Complete (FCC) is influenced by a host of other variables. Each one of these shadow variables can be found in another view of the model - "Effect of Experience on FCC", for example, can be found in the "Staffing and Experience" model view, as presented in section 5.2.8. In this case we are "linking" this view to other parts of the model using a formula for *Fraction Correct and Complete* that incorporates all these shadow variables to compute a (dynamically changing) FCC value.

It is important to note that there is a distinction to be made between the model itself, and a given simulation (or "run") of the model with specific parameters. The model does not change, but it can be run with different parameters.

5.2 Modeling the Seven Genes of Agile

The following section describes sections of the Agile Project Dynamics (APD) model pertaining to agile methods, and delves into the specifics of how each of the seven "agile genes" characteristics is modeled in its own view. Where possible, Scrum is used as an example and as the "reference methodology" when modeling certain aspects of Agile development.

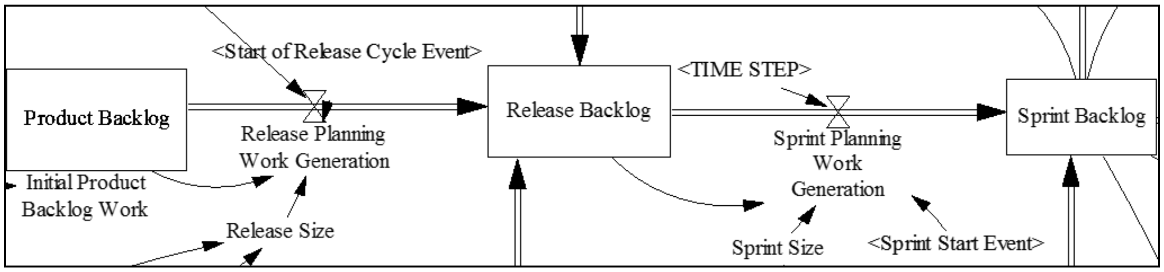
5.2.1 Story/Feature Driven

We begin to model the Story/Feature driven nature of an Agile project by modifying the structure of the rework cycle to account for the stocks (or "buckets") of features that are maintained as software capabilities are envisioned, planned, prioritized, developed, tested, and reworked.

This is better understood when considering that in Scrum, the following stocks of work exist:

- *Product Backlog*: A backlog exists and evolves over the lifetime of the product; it is the product road map. At any point, the Product Backlog is the single, definitive view of "everything that could be done by the Team ever, in order of priority". The Product Backlog is continuously updated to reflect changes in the needs of the customer, new ideas or insights, moves by the competition, technical hurdles that appear, and so forth. (Deemer et al. 2009)
- *Release Backlog*: The subset of the Product Backlog that is intended for the upcoming release of the software. This is the result of planning and prioritization to select which features in the product backlog need to be implemented in the next cycle.
- *Sprint Backlog*: The list of tasks/work that the development team must address during the next sprint. It consists of a set of tasks required to complete the features selected for a subset of the Release.

Note that the features in the *Product Backlog* and *Release Backlog*, in Scrum, would typically be sized using "story points," an estimate of the effort that will be required to complete them, whereas the Sprint Backlog is a collection of tasks. Therefore all of these backlogs can be modeled as stocks of fungible "tasks" in our APD model. These structures of the model are shown in Figure 40:



<Start of Release Cycle Event>
Product Backlog
Release Planning Work Generation
Initial Product Backlog Work
Release Size
Release Backlog
<TIME STEP>
Sprint Planning Work Generation
Sprint Size
<Sprint Start Event>
Sprint Backlog

Figure 40 – SD Model structure of the Backlogs in an Agile Project

A project begins with an initial amount of work (in unit of "tasks") in the *Product Backlog*. A subset of those tasks is moved into the *Release Backlog* at the start of a release cycle. Each sprint in turn takes a subset of the *Release Backlog* into the *Sprint Backlog* to begin development work.

Work within a sprint follows the dynamics of the Rework Cycle, described in section 4.2. The sprint rework cycle in our APD model is shown below in Figure 41 - Sprint Rework Cycle

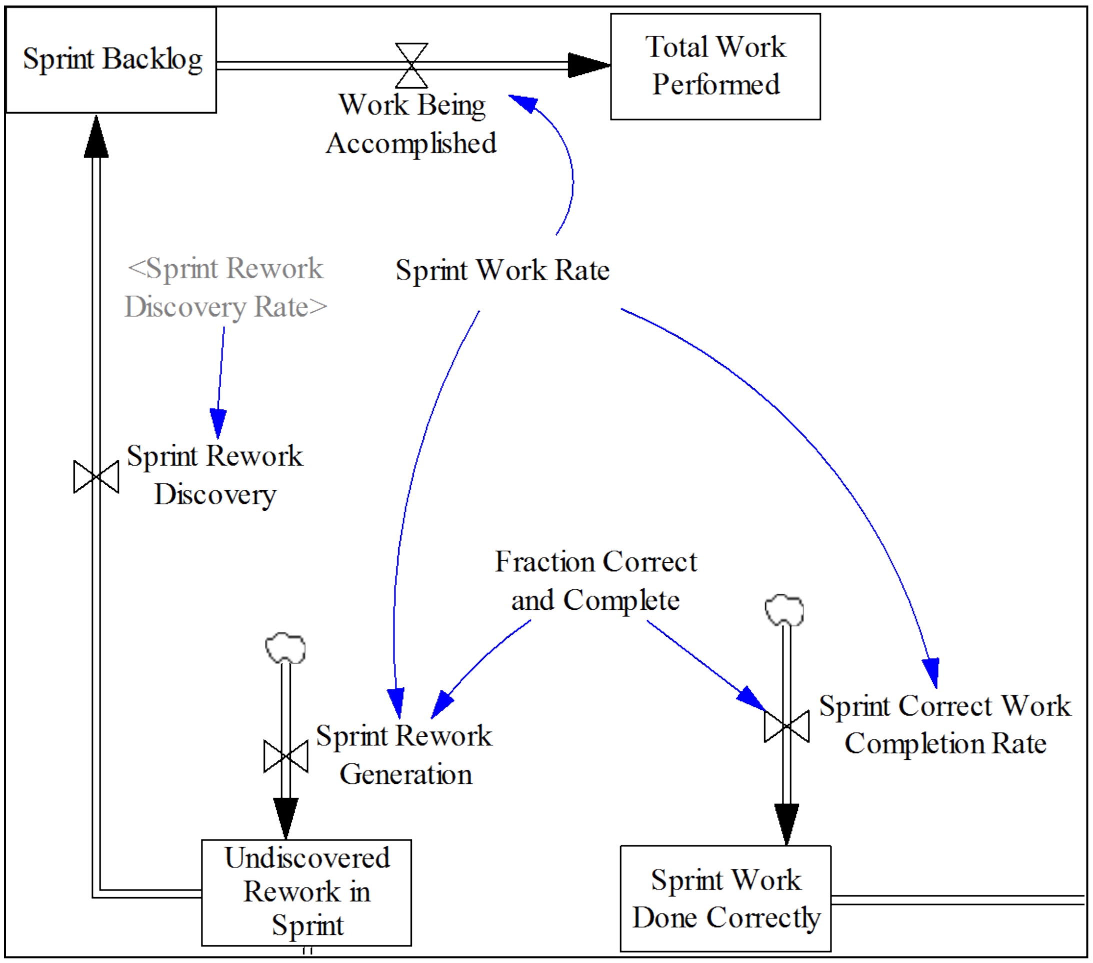

Sprint Backlog
Total Work Performed
Work Being Accomplished
<Sprint Rework Discovery Rate>
Sprint Work Rate
Sprint Rework Discovery
Fraction Correct and Complete
Sprint Rework Generation
Sprint Correct Work Completion Rate
Undiscovered Rework in Sprint
Sprint Work Done Correctly

Figure 41 - Sprint Rework Cycle

Also, as part of this gene, we assume that the recipient of the software product is willing to accept the delivery of software releases as integrated "feature sets" or sets of client-valued functionality.

5.2.2 Iterative-Incremental

To model the iterative and incremental nature of an Agile project, we must repeat the process described above in several builds, within several releases, and with a fluctuating *Product Backlog*. Since, as described above, the transition of tasks (quantities of work) from one backlog to the other is time/event based, we need signals indicating the start and end of release and sprint cycles, to trigger the transition of work.

Figure 42 shows the model structure used to generate the *Start of Release Cycle Event* and the *End of Release Cycle Event* signals observed in Figure 43. The number of *Planned Releases* and the length of the project (equal to FINAL TIME) are used to calculate the length of a release cycle. "*InRelease*" is a control flag that is helpful in other parts of the model, used in a Boolean fashion to indicate if we are currently within a release cycle or in between cycles. In our model we have set the project length to 104 weeks (*FINAL TIME* = 104 weeks), with 4 planned releases (*Number of Releases if Agile* = 4), and half planning week in between releases (*Release Planning Duration* = 0.5 weeks).

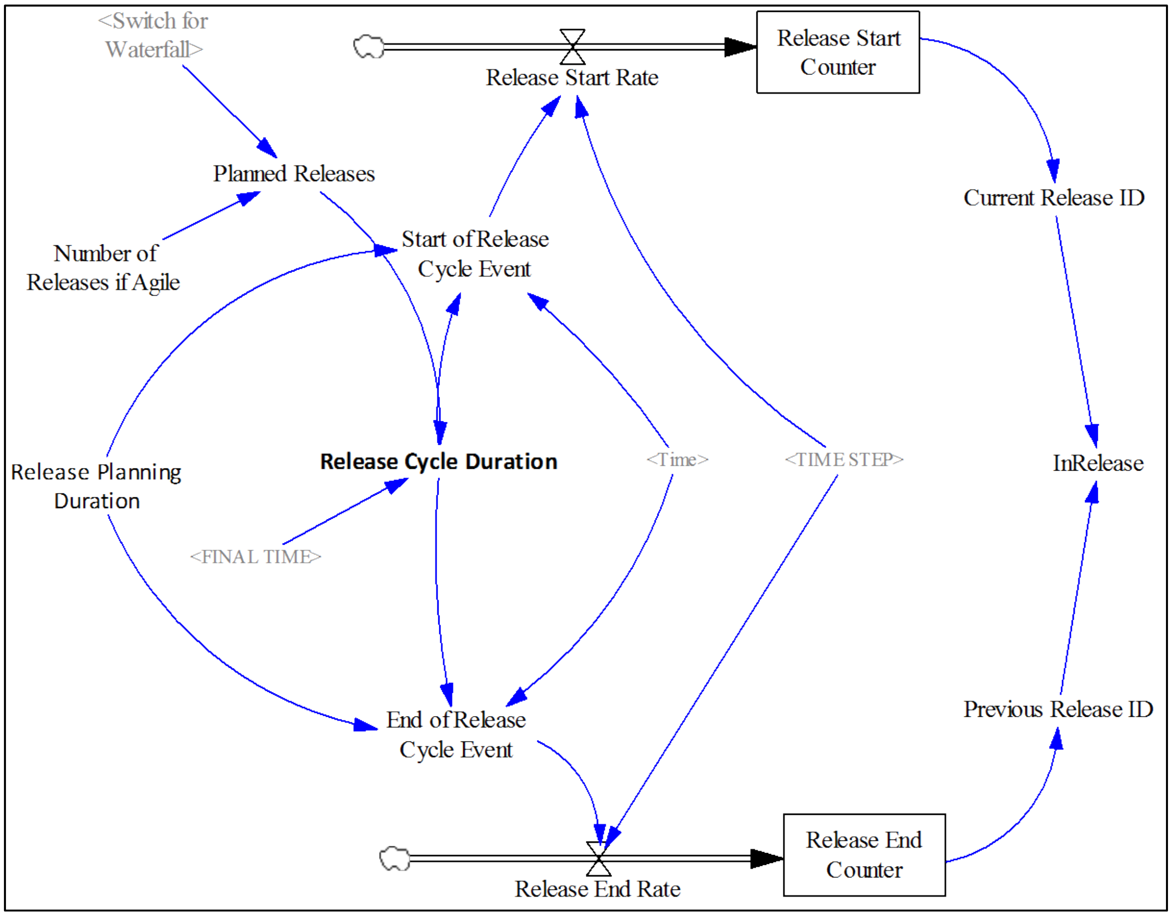


Figure 42 - Release Cycle Timing Model Elements

To support the behavior of a waterfall project, we also introduce the *Switch for Waterfall* control, which if set, causes the *Release Cycle Duration* to equal the duration of the entire project (this simulates a single-pass waterfall development approach.) We can thus generate events to drive a multi-release or a single pass development lifecycle by simply switching on/off this control.

With the parameters described above, and with the waterfall switch set to "off", this results in a 26 month *Release Cycle Duration* as can been seen in Figure 43. On the other hand, turning on the *Switch for Waterfall* results in the single-pass release cycle shown in Figure 44.

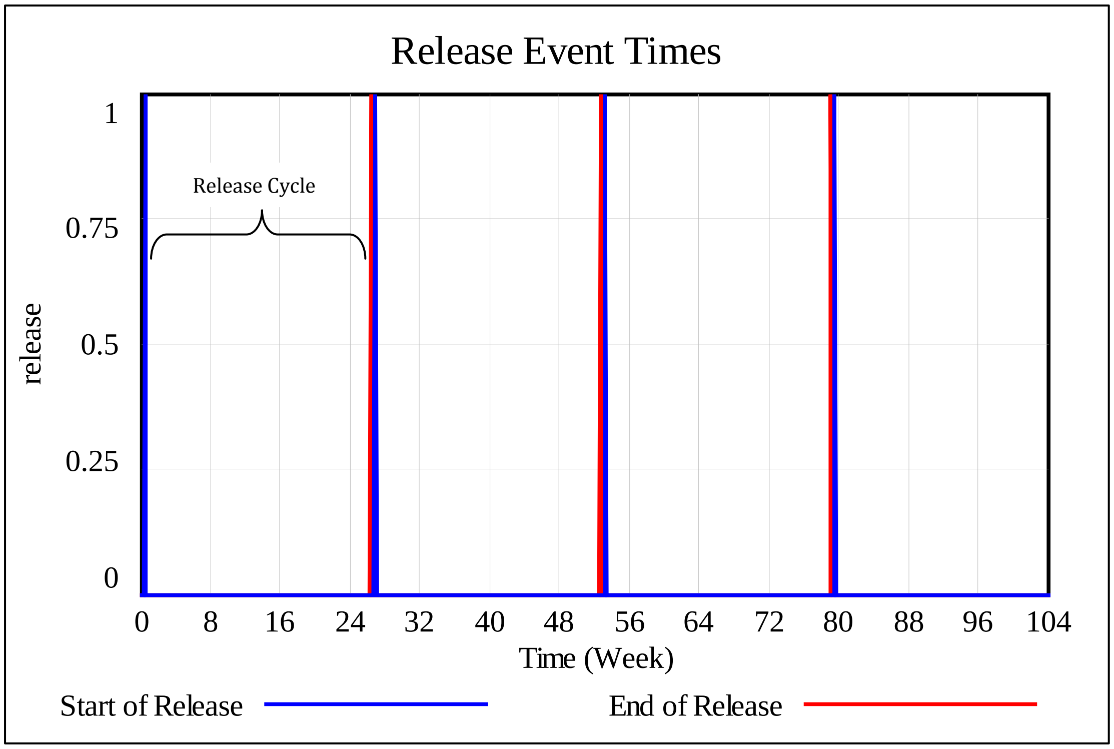


Figure 43 - Release Event Times (Multi-release)

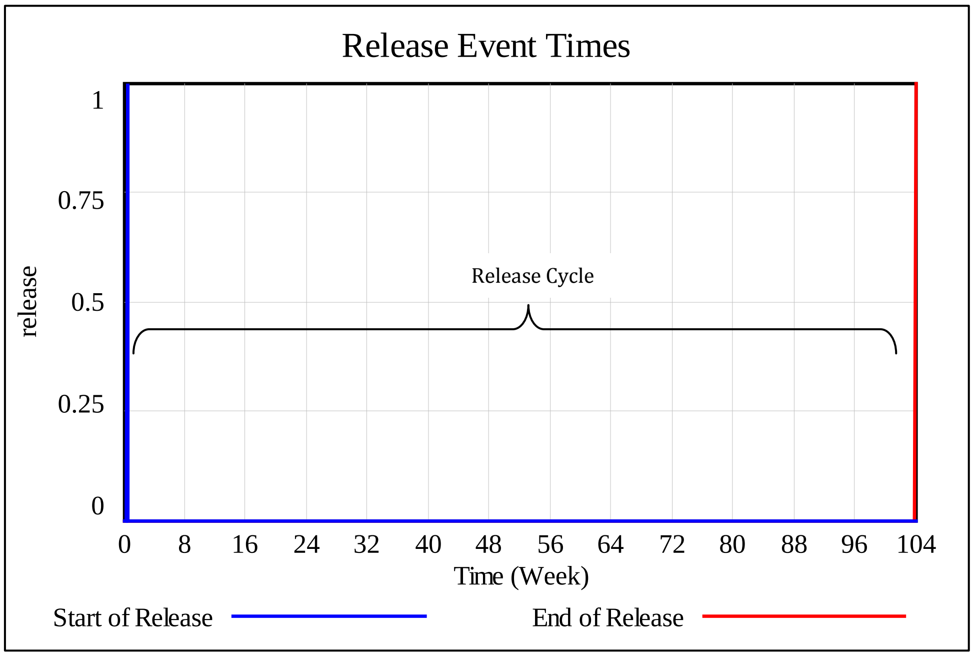


Figure 44 - Release Event Times (Single Release)

A similar structure (seen in Figure 45) is used to generate sprint start and end events (seen in Figure 46):

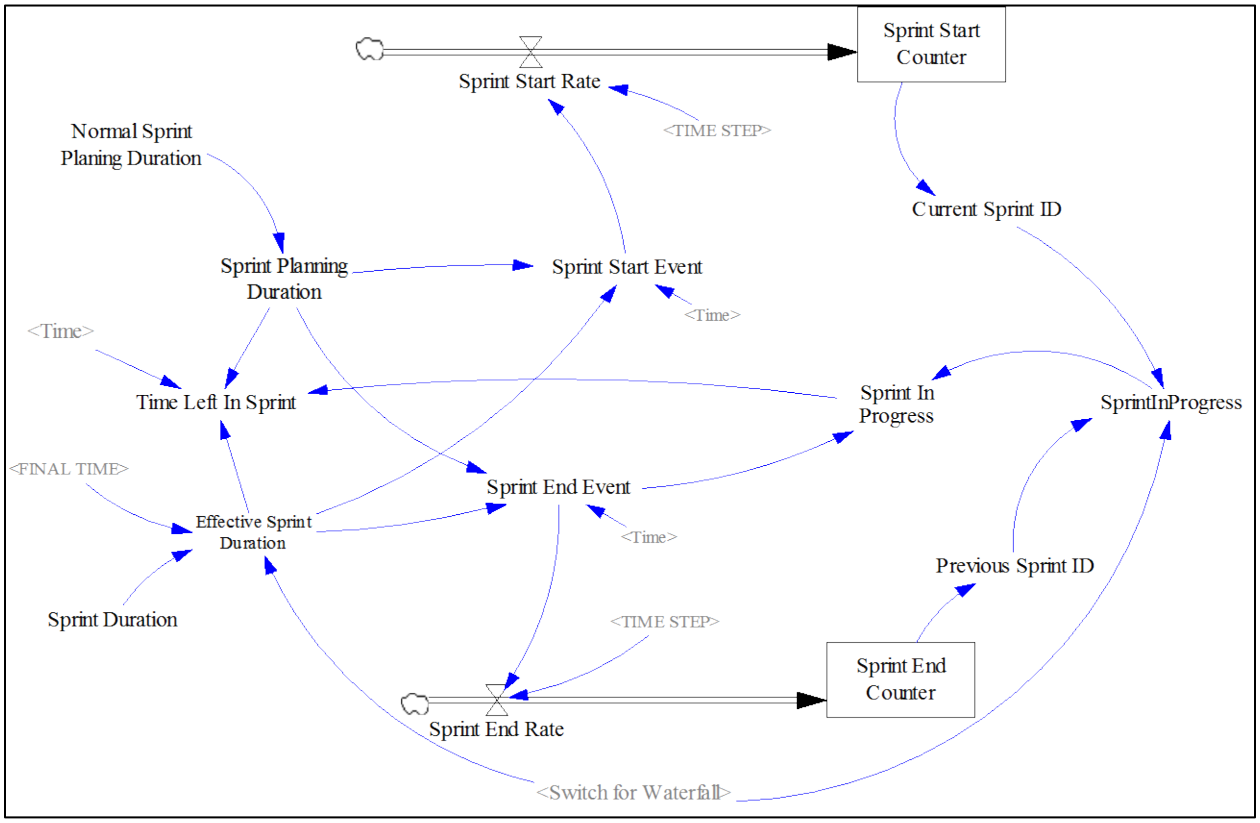


Figure 45 - Modeling of Sprint-related events

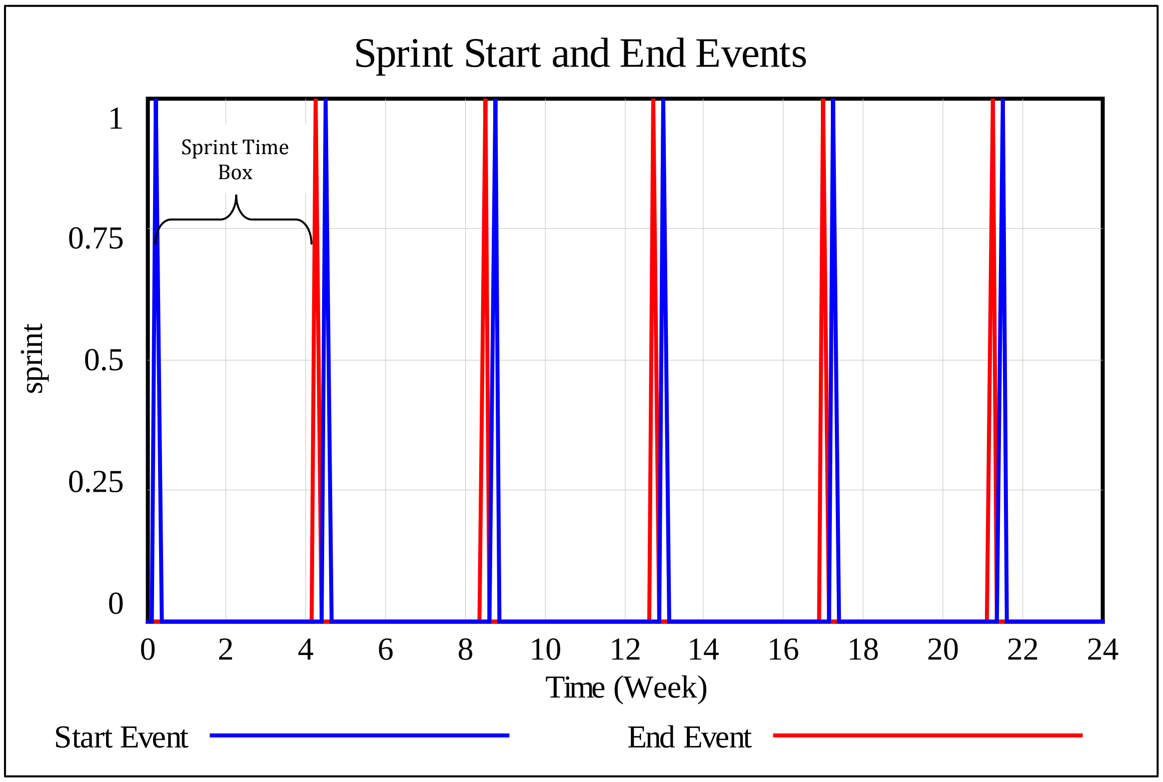


Figure 46 - Sprint Start and End Times

If we execute our model with all of the above, while monitoring the flow of work in our three backlogs, we observe the behavior shown in Figure 47: at the start of a release (*Start of Release Cycle Event*), a set of tasks (*Release Size*) is moved from the *Product Backlog* to the *Release Backlog*. At the start of a sprint (*Sprint Start Event),* an amount of work (*Sprint Size*) is moved from the *Release Backlog* to the *Sprint Backlog*. As work is performed, tasks are consumed from the *Sprint Backlog*.

In Figure 47, the top line represents the *Product Backlog:* we observe it decreasing at the beginning of every Release The middle line is the release backlog, which decreases by small amounts at frequent intervals (the *Sprint Duration*). Finally, the almost imperceptible (at this scale) line at the bottom represents tasks in the *Sprint Backlog.*

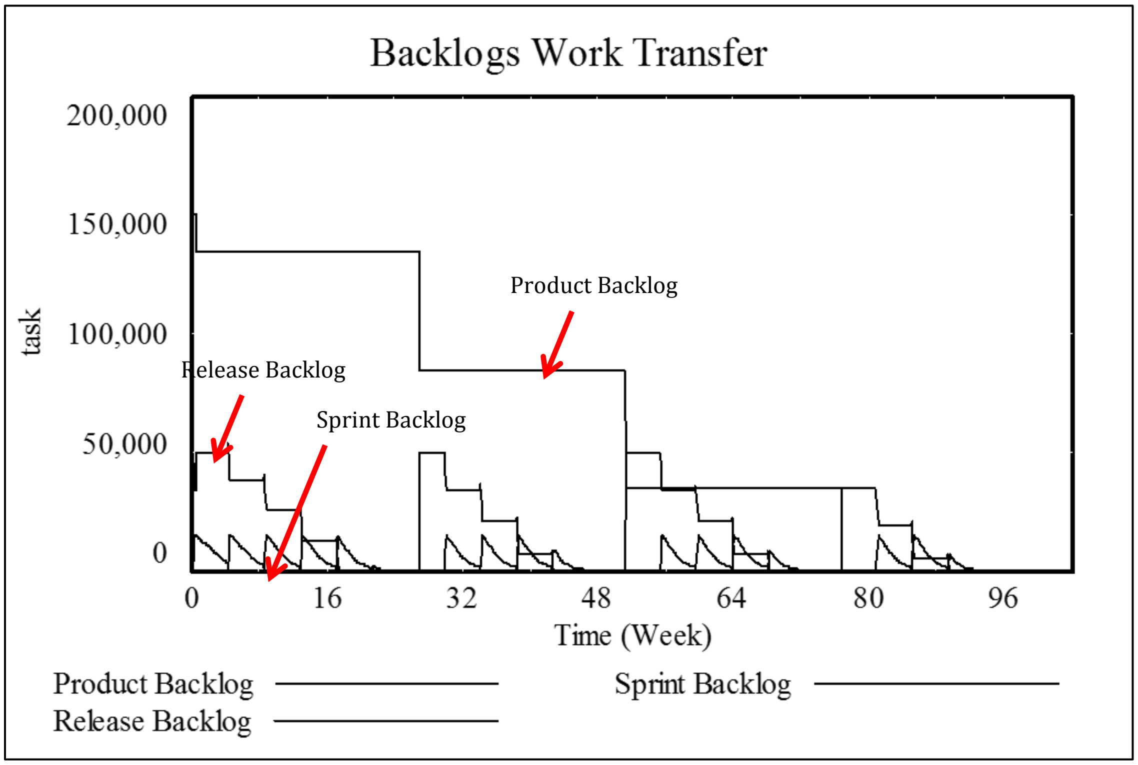


Figure 47 - Transfer of Work through Backlogs

To emulate a single-pass waterfall project, we employ the *Switch for Waterfall* control to simply set the sizes of these three backlogs to an equal value, thus executing one big one big release with a set feature list. The effect of this can be seen below in Figure 48.

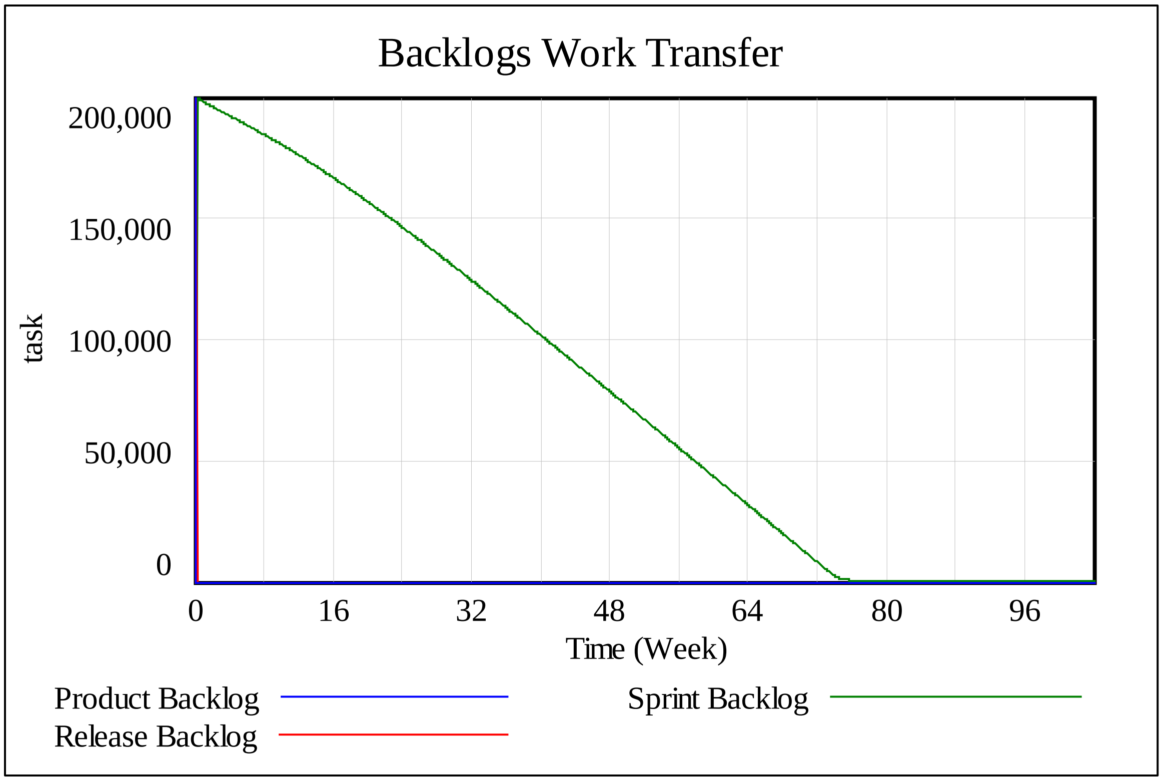


Figure 48 - APD Model Emulating a Single-pass Waterfall Project

Here, having ‘turned on’ the *Switch for Waterfall*, we observe that the product and release backlogs immediately drop to zero, as all of the development work is being done in one giant sprint, effectively emulating the performance of a waterfall-style project.

5.2.3 Refactoring

As described in section 3.3.3, refactoring is the work required to restructure the software baseline in order to pay off the “Technical Debt.” Technical Debt can be modeled as a stock representing an accumulation of tasks over time that, once they reach a certain threshold, must be accomplished before any other development can proceed.

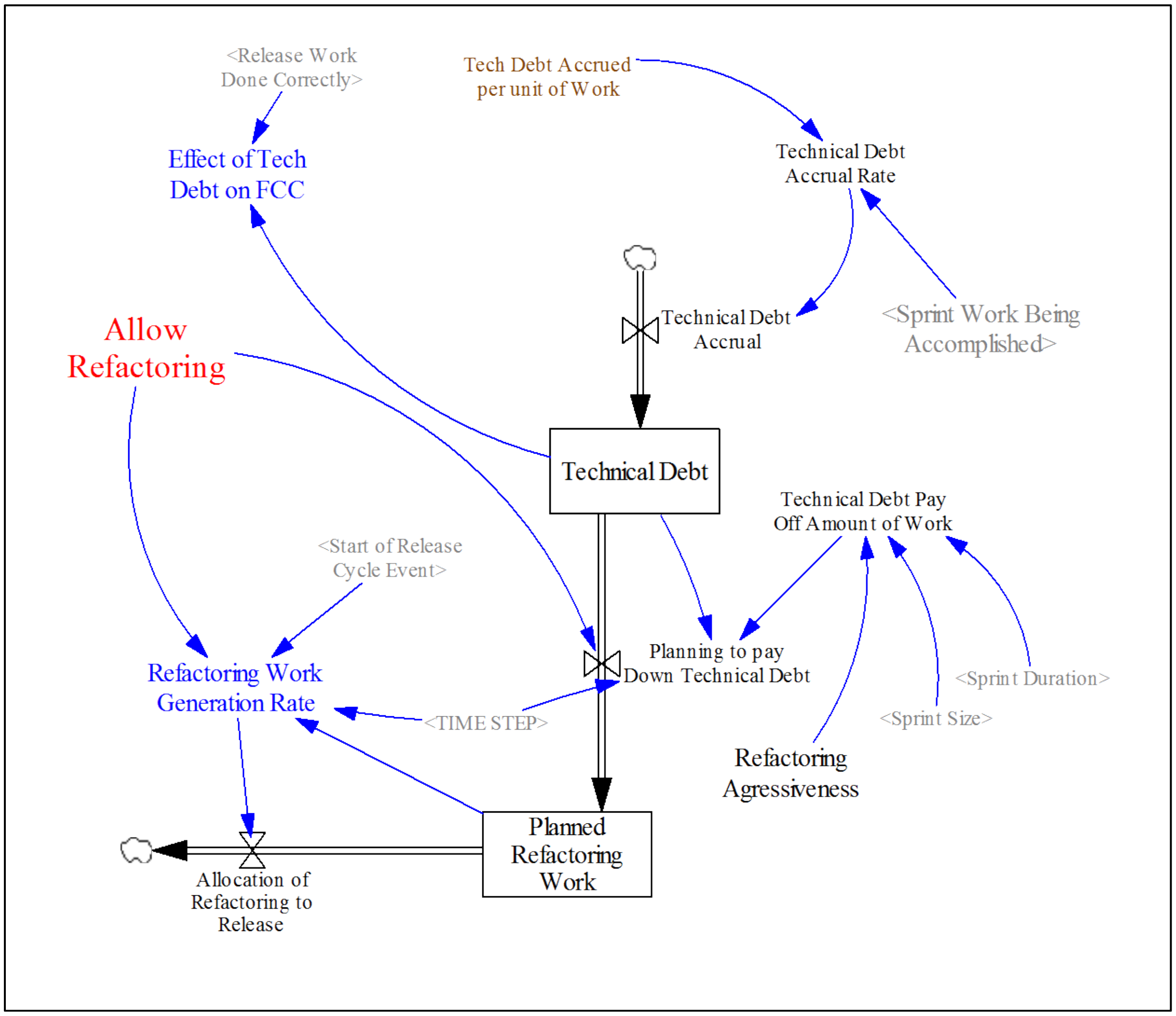

<Release Work Done Correctly>
Tech Debt Accrued per unit of Work
Effect of Tech Debt on FCC
Technical Debt Accrual Rate
Technical Debt Accrual
<Sprint Work Being Accomplished>
Allow Refactoring
Technical Debt
Technical Debt Pay Off Amount of Work
<Start of Release Cycle Event>
Planning to pay Down Technical Debt
Refactoring Work Generation Rate
<TIME STEP>
<Sprint Duration>
<Sprint Size>
Refactoring Agressiveness
Planned Refactoring Work
Allocation of Refactoring to Release

Figure 49 - Model Elements for Refactoring

Figure 49 shows the APD model structure elements and feedbacks that produce this dynamic. As work in performed in the project, *Technical Debt* accrues at the *Technical Debt Accrual Rate*. *Tech Debt Accrued per unit of Work* represents the percentage of each completed work task that is susceptible to refactoring at a later point in time. Quantifying the *Technical Debt Accrual Rate* is extremely difficult; it cannot be calculated a priori, especially for a legacy program where development may occur "on top of" an existing baseline with an unknown technical debt quantity.

One way to derive a measure for technical debt is to assess the "quality" of one's code. The SQALE (Software Quality Assessment based on Lifecycle Expectations) is one model for doing this. It is a methodology for assessing the quality of code, using algorithms to score software quality along the dimensions of: Reusability, Maintainability, Efficiency, Changeability, Reliability, and Testability. This type of analysis is available in several static analysis packages including: Insite SaaS, Sonar, SQuORE, and Mia-Quality.[12] Some have been using the SQALE SQI (Software Quality Index) as a measure of technical debt.

[12] http://www.sqale.org/tools

Agile practitioners suggest other, simpler approaches: Agile teams can monitor the amount of refactoring compared to the amount of new feature work in each sprint or iteration to establish a historical baseline for *Technical Debt Accrual*.

For testing purposes of our APD model, we will measure technical debt as a fraction of the amount of correct work already performed (*Release Work Done Correctly*). The reasoning behind this is that *Release Work Done Correctly* is proportional to the size of the code base (i.e. SLOC.) It also follows that the more SLOC, the more technical debt, as it has been long established that lines of code are correlated to effort and defects, and that duplication of code is by far the most common form of technical debt.[13] It's also been highlighted by mantras in a couple of books: the DRY (don't repeat yourself) principle in the *Pragmatic Programmer* (A. Hunt, and D. Thomas, Addison Wesley, 1999) and "Once and Only Once" from *Extreme Programming Explained: Embrace Change* (K. Beck, Addison Wesley, 1999). (Fowler, 2001)

In our model we will use a value 0.05 to represent 5 units of technical debt for every 100 correct tasks completed to drive *Technical Debt Accrual*. If Refactoring is practiced (using the switch *Allow Refactoring*) Once the *Technical Debt* level reaches the *Technical Debt Pay Off Amount of Work*, then that amount of work is moved into *Planned Refactoring Work* to be performed in the next sprint. Figure 50 shows the accumulation of *Technical Debt* over time. Once it reaches *Technical Debt Pay Off Amount of Work*, the stock is drained as the work is planned into the next release.

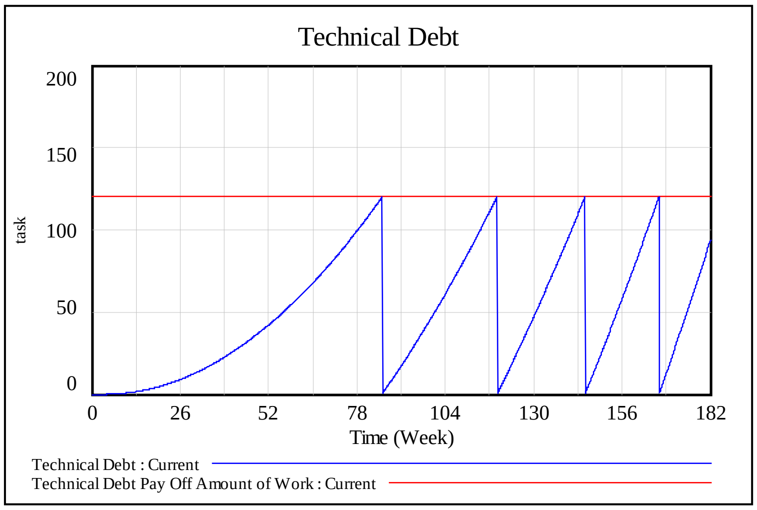


Figure 50 - Accumulation and Pay Off of Technical Debt

[13] http://jamesshore.com/Blog/An-Approximate-Measure-of-Technical-Debt.html

5.2.4 Micro-Optimizing

As described in section 3.3.4, the Micro-Optimizing gene represents the adaptive nature of a given project's development processes. In an Agile project, we can model this by recognizing that at the end of each iteration, the team tweaks and fine tunes the process to gain small gains in Productivity, FCC, and a small improvement in Rework Discovery. Additionally, the team learns over time what their work capacity is, and they dynamically adjust the size of each sprint by reducing or increasing the amount of work they choose to tackle in the next sprint, based on performance in a previous sprint.

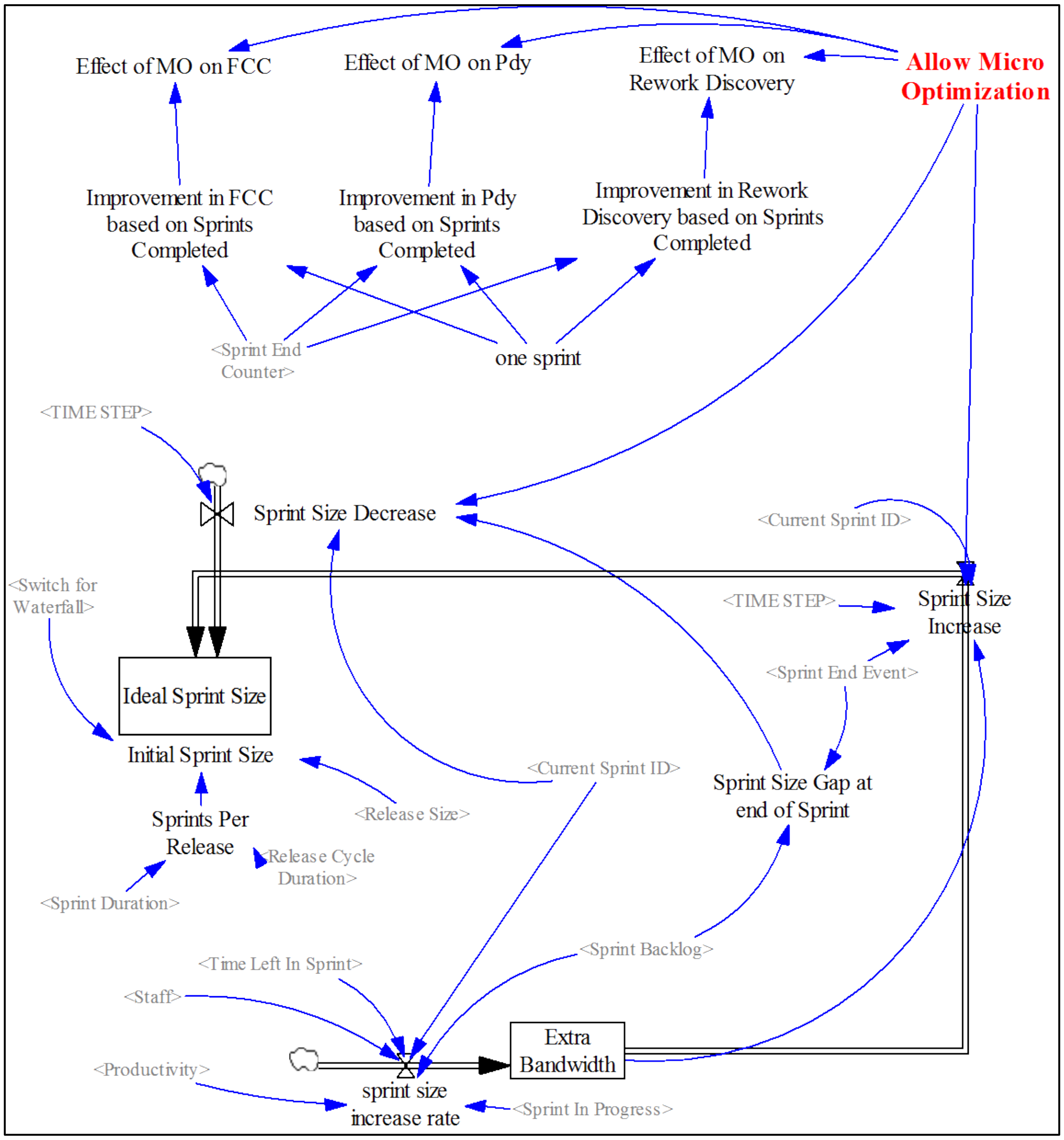


Figure 51 - APD Model Elements for Micro-Optimization

Figure 51 shows how we add this to our APD model. At the end of each iteration, if any work is remaining in the *Sprint Backlog*, it represents the "gap" between the size of the previous sprint and the amount of work that the team was able to accomplish. This gap is represented by the variable *Sprint Size Gap at end of Sprint*. This gap amount is used to decrease *Ideal Sprint Size*, at the next *Sprint End Event. Ideal Sprint Size* will be used to set the variable controlling the size of the next sprint, *Sprint Size*.

On the other hand, if the team finishes all of the *Sprint Backlog* work before the end of the sprint, the *Time Left In Sprint* at that point is used to determine how much *Extra Bandwidth* the team had to spare in that sprint, and that amount will be used to increase the size of the next sprint via the *Sprint Size Increase* flow.

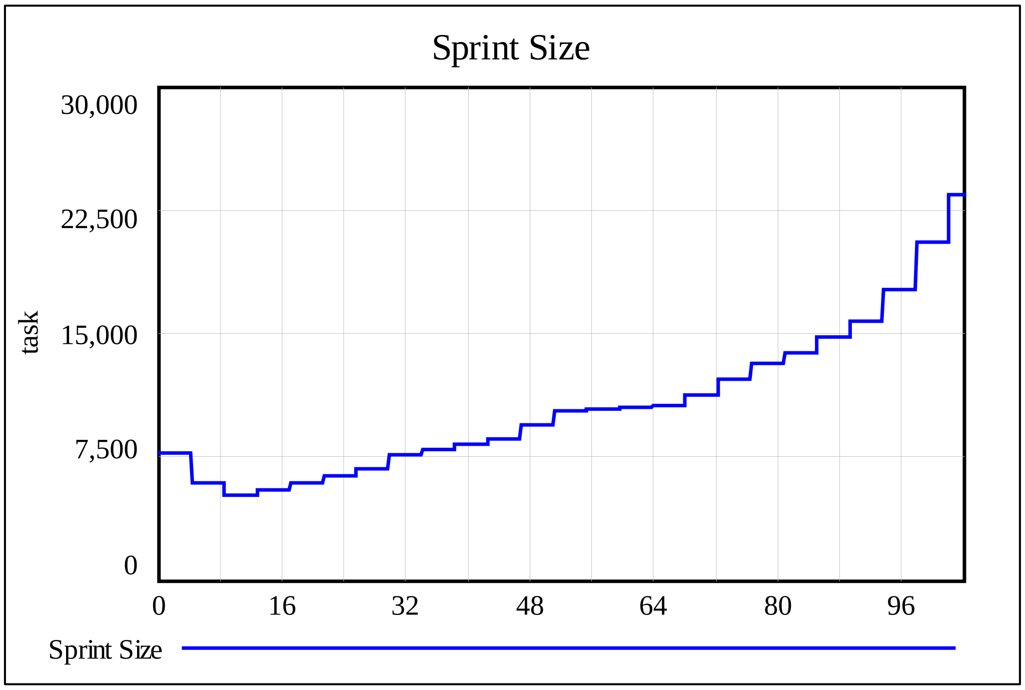


Figure 52 - Example of Dynamically Changing Sprint Size

Figure 52 above shows the behavior produced when we turn on the control *Allow Micro-Optimization*: a dynamically changing *Sprint Size* over time when running this model. We interpret this graph as follows: the team selects an initial sprint size of 7500 tasks, based on initial project parameters (*Sprint Duration*, *Number of Releases,* and *Release Size*). Then, as development work proceeds, the team learns and adapts while dynamically changing the sprint size, which represents how much work the team can handle within a single sprint. At first, there is a dip in this capacity, as the project is still assimilating its' new inexperienced staff, and while requirements are still uncertain. As the project progresses, the team becomes more and more productive, while generating less defects, allowing them to bite off larger amounts of work as the project proceeds. This behavior matches what is observed in industry: Scrum teams report that after a dozen or more sprints they become "fine-tuned" and capable of tackling more work per sprint.

The other set of effects that we have gathered under the micro-optimization genes, as discussed earlier, are its effects on Fraction Correct and Complete, on Rework Discovery Time, and on Productivity. In its current form the APD model uses a simple approach for modeling these simply as a function of the number of sprints completed. In other words, the more sprints, the better the team performs along these dimensions. Quantifying this however is a difficult task, much like quantifying technical debt. A good approach here would be to derive these numbers from historical performance data. For the purposes of our modeling, we use a simple lookup table with very conservative values. This produces the following improvement over time for FCC, for example as seen in Figure 53.

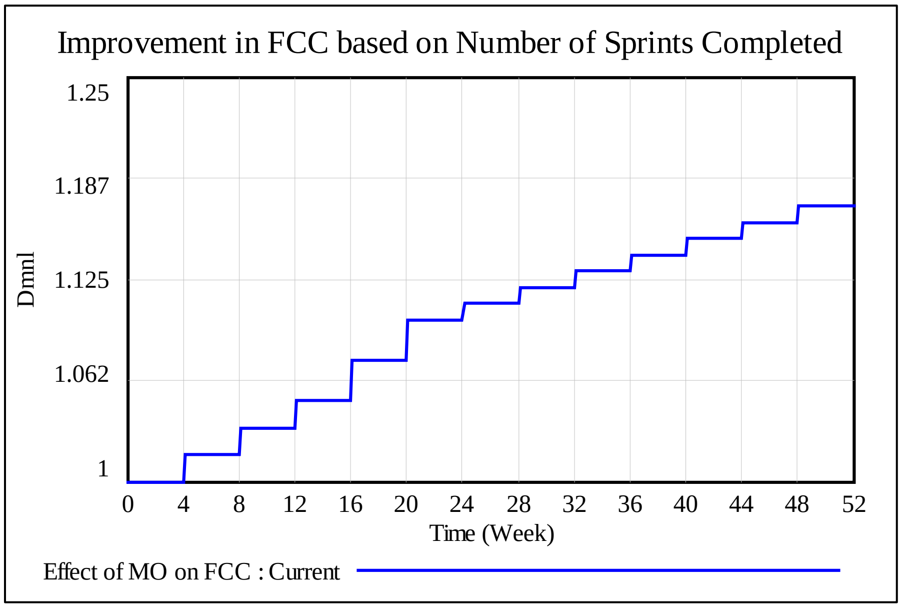


Figure 53 - Example Model Behavior: FCC Improvement as a Function of Sprints Completed

5.2.5 Customer Involvement

As detailed in section 3.3.5, Customer Involvement has some positive and some negative effects on our core APD rework cycle. What could be considered a negative effect is the fact that continuous customer input results in a certain level of requirements churn. We model this (Figure 54) by calculating the *Fraction of Release Work Completed.* Based on this measure of work progress, we employ a lookup table in *Effect of Customer Involvement on Requirements Churn.* This variable represents the percentage of the requirements that will change based on customer input and how far along in the release we are. For the purposes of this model we are using a bell-curve reference mode, with a maximum of 10% churn. The rationale behind this is that the most churn occurs toward the middle of the release as tangible software is produced and allows the customer to feedback changes into the release.

The positive effects of Customer Involvement are captured in *Effect of Customer Involvement on FCC* and *Effect of Customer Involvement on Rework Discovery Time*.

Effect of Customer Involvement on FCC: A large part of software defects are caused by requirements uncertainty. As progress is made during each sprint, requirements uncertainty is eliminated with customer feedback and product demos. This in turn improves our FCC.

Effect of Customer Involvement on Rework Discovery Time: Likewise, customer availability to answer questions, detect conflicts and misunderstandings , and to identify usability issues during product demonstrations means that we are more likely to identify rework early in the process.

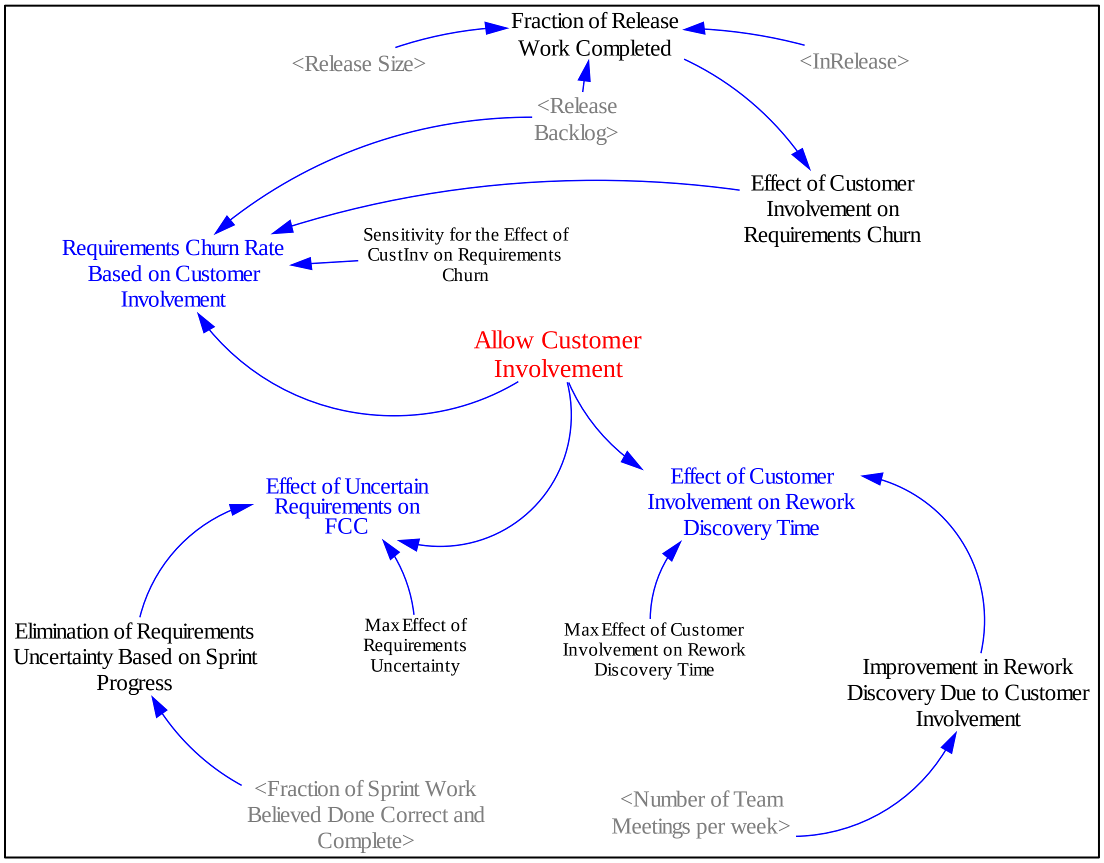

Fraction of Release Work Completed
<Release Size>
<InRelease>
<Release Backlog>
Effect of Customer Involvement on Requirements Churn
Requirements Churn Rate Based on Customer Involvement
Sensitivity for the Effect of CustInv on Requirements Churn
Allow Customer Involvement
Effect of Uncertain Requirements on FCC
Effect of Customer Involvement on Rework Discovery Time
Elimination of Requirements Uncertainty Based on Sprint Progress
Max Effect of Requirements Uncertainty
Max Effect of Customer Involvement on Rework Discovery Time
Improvement in Rework Discovery Due to Customer Involvement
<Fraction of Sprint Work Believed Done Correct and Complete>
<Number of Team Meetings per week>

Figure 54 - APD Model Elements for Customer Involvement

In Figure 55, we see the *Requirements Churn Rate Based on Customer Involvement*. This value fluctuates throughout each release between 0 and 30 tasks per week, in this example. In this simulation, we have four software releases, with three sprints in each release. Notice how, in each release, the requirements churn rate has decreasing several steps. These coincide with each sprint within a release – this behavior is intended to represent the notion that at the beginning of a release requirements are more fluid and subject to change, but as more sprints are completed and as there are less remaining features and work to do in the current release, there are fewer requirements subject to churn.

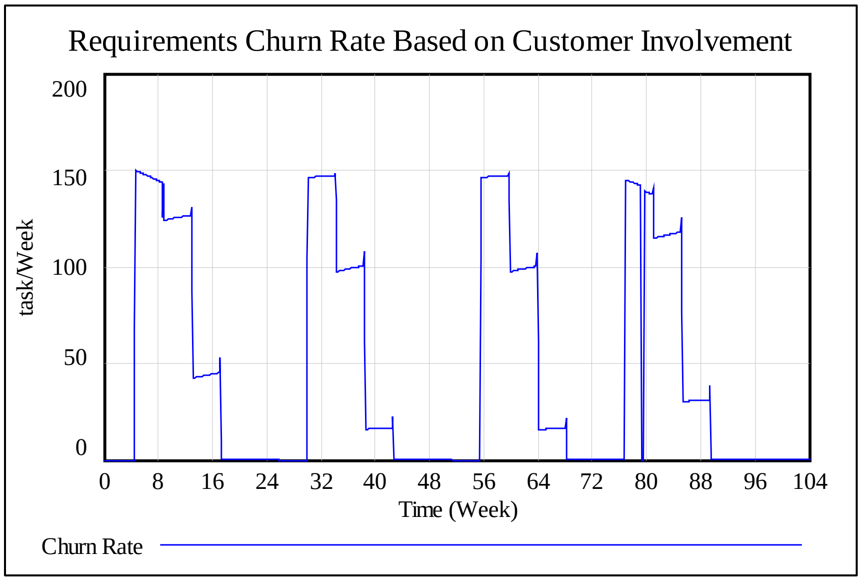


Figure 55 - Example of Requirements Churn due to Customer Involvement

Below in Figure 56 we see also the *Effect of Customer Involvement on FCC*. As each sprint nears completion, the effect of having a customer available to validate work products results in an improvement in FCC, as uncertainty is reduced. Our model places a lower limit of 0.85 on this effect (*Max Effect of Requirements Uncertainty)* meaning that at worse, requirements uncertainty can reduce the FCC by 15%. In this graph we see the value for this effect climb up near to 1, then drop back down to 0.85 at regular intervals, coinciding with the sprints in the project. This behavior is intended to represent the fact that, as each sprint nears completion, the requirements for the features under development in this sprint are less and less unclear (especially with customer involvement in scrums and feature demos).

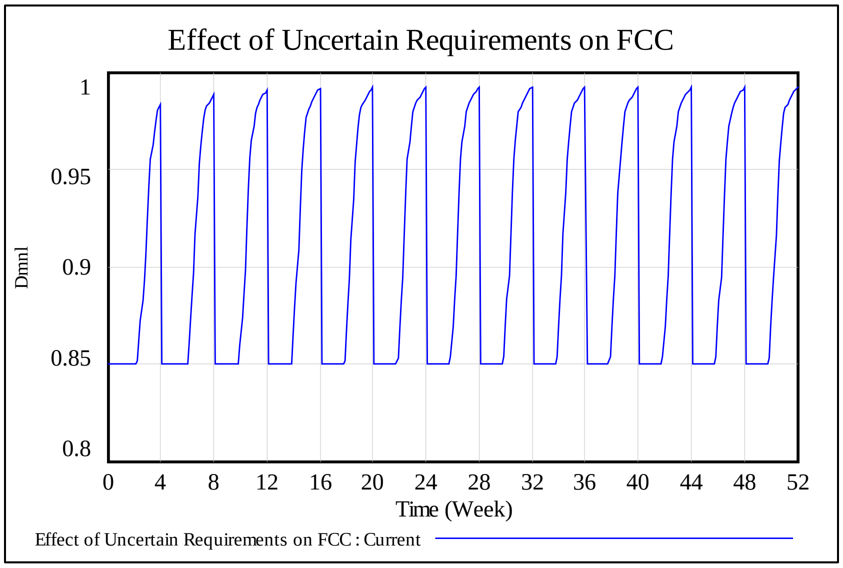
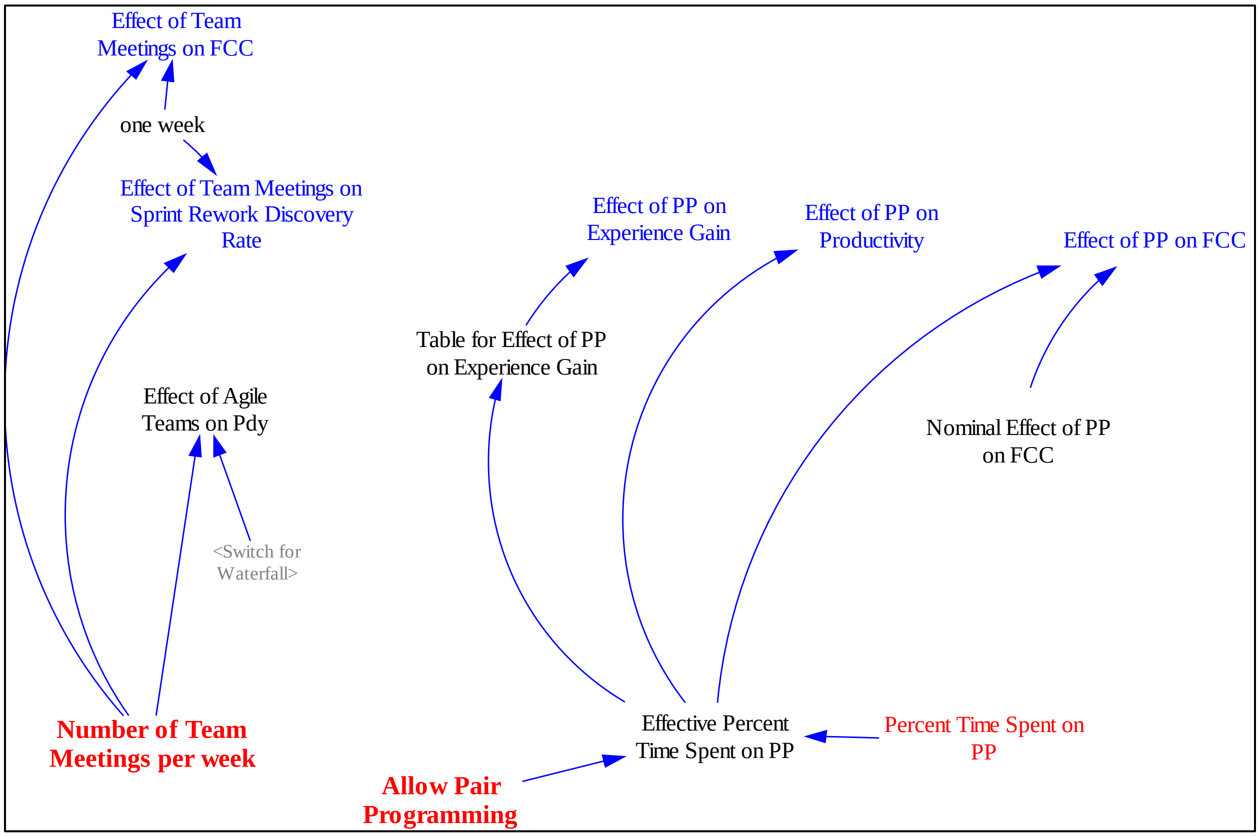


Figure 56 - APD Example Effect of Customer Involvement on FCC

5.2.6 Team Dynamics

The portion of the model, shown in Figure 57, is a very simplistic approximation of the effects of the practices of Pair Programming (PP), and frequent (e.g. daily) meetings.



Effect of Team Meetings on FCC
one week
Effect of Team Meetings on Sprint Rework Discovery Rate
Effect of Agile Teams on Pdy
<Switch for Waterfall>
Number of Team Meetings per week
Effect of PP on Experience Gain
Table for Effect of PP on Experience Gain
Effect of PP on Productivity
Effect of PP on FCC
Nominal Effect of PP on FCC
Effective Percent Time Spent on PP
Percent Time Spent on PP
Allow Pair Programming

Figure 57 - APD Model Elements for Team Dynamics

We know that in industry pair programming is not practiced 100% of the time. Typically developers spend 20% to 50% of their time doing PP. Thus *Percent Time Spent on PP* is set to 0.5 for our model, however it can be adjusted per project conditions.

We employ a switch (*Allow Pair Programming*) in the model to control whether Pair Programming is practiced - we can thus model the effects of PP on other parts of the system. These are:

- *Effect of PP on Experience Gain*: studies have shown greater experience gain through PP. We capture this effect in this variable.
- *Effect of PP on Productivity*; PP is reported to cause a reduction in productivity measures such as SLOC-per-month-per-person.
- *Effect of PP on Defect Generation*; PP results in better design and less defects, as it creates a form of real-time code review.

-

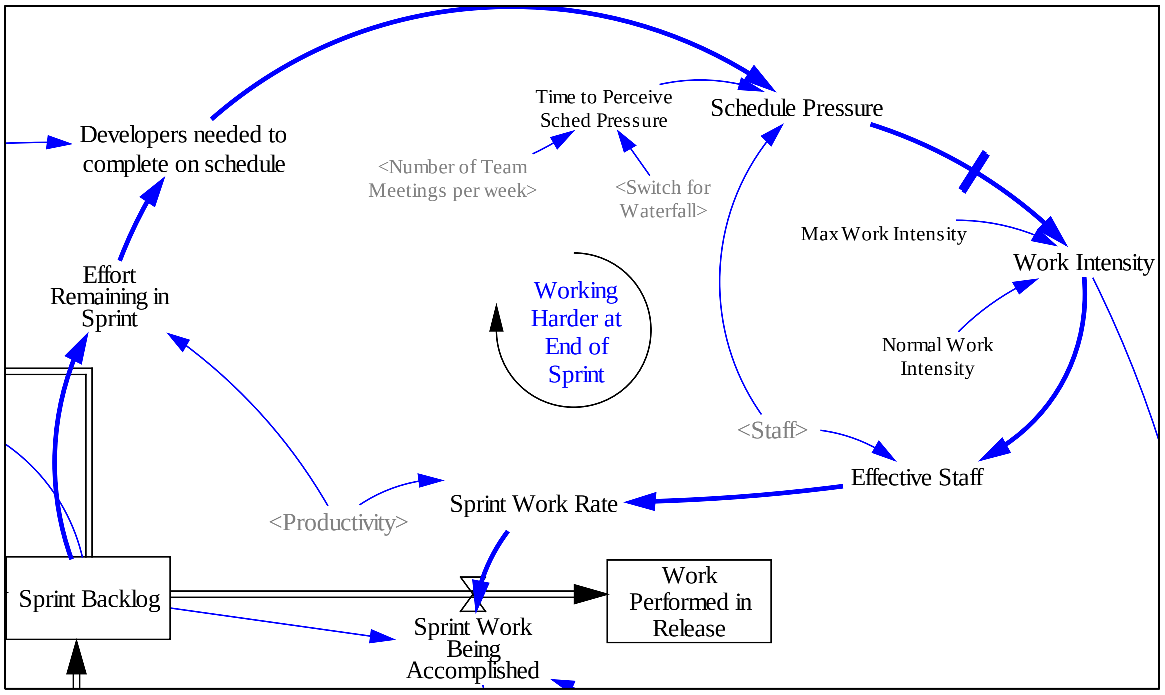
We also employ a variable "*Number of Team Meetings per week*" to simulate Frequent team meetings (e.g. daily Scrum) and the two effects these have on the system:

- *Effect of Team Meetings on FCC*: The more frequent meetings, the more issues are raised, questions answered, leading to less defects in the software.
- *Effect of Team Meetings on Sprint Rework Discovery Rate*: Rework is discovered sooner through constant feedback, especially if a customer is involved in these meetings.

There is a third (and negative) effect that could be associated with the practice of regular meetings. Is it not true that the more meetings, the less time spent doing work? In most cases, the answer is yes. However in an Agile world, especially in the Scrum process, great attention is paid to making these team meetings extremely short and extremely productive. Daily Scrums are ideally ten to fifteen minutes stand-up meetings, often scheduled in the morning to start the day. The time this takes away from 'doing the work' is therefore negligible, which is we haven't modeled this aspect.

The other aspect of Team Dynamics that we presented in section 3.3.6 relates to the nature of schedule pressure in the Agile environment: frequent short bursts of schedule pressure are experienced at the end of each sprint.

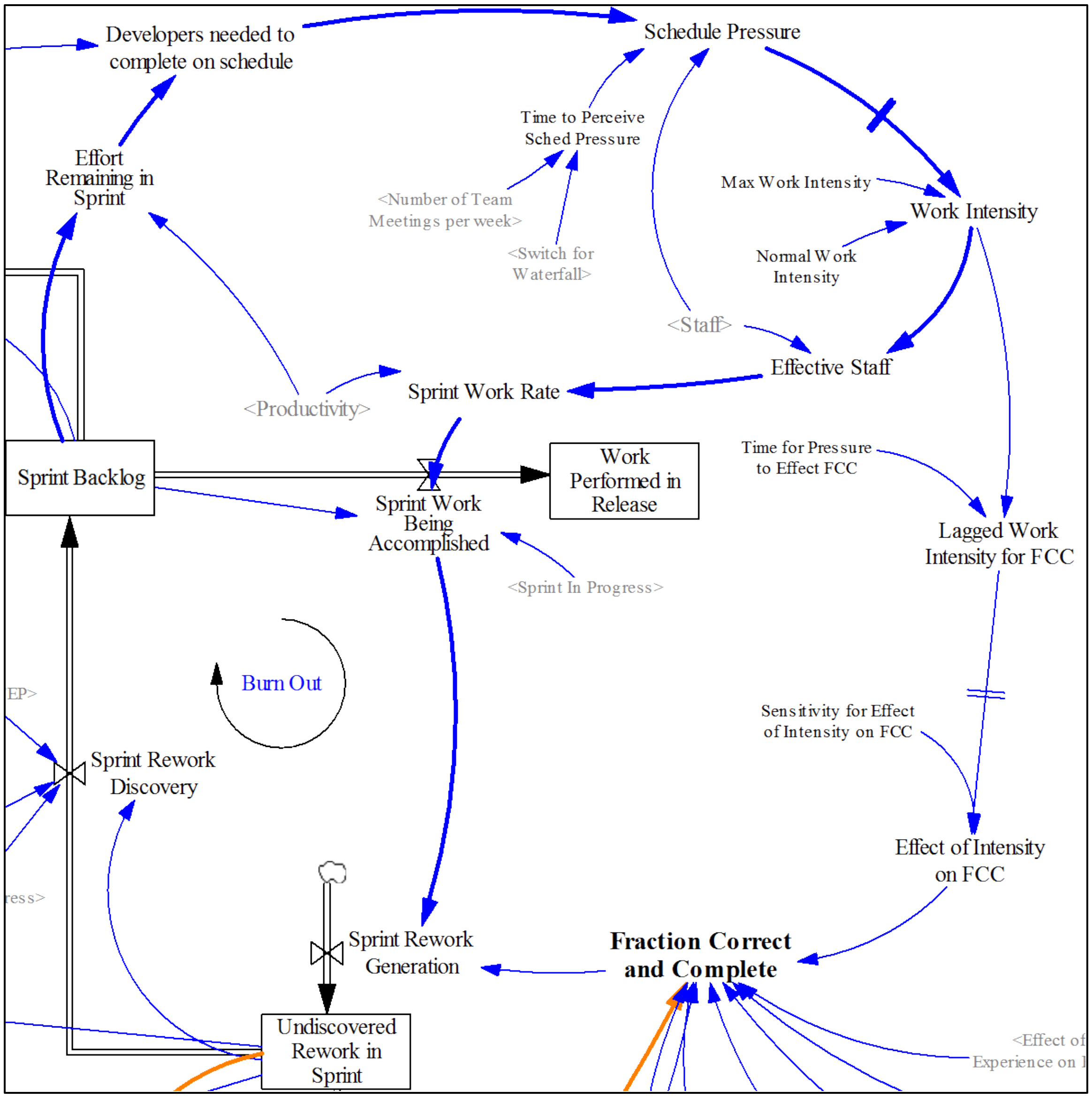
This dynamic is captured in the feedback loop shown in Figure 58. A calculation of the number of *extra Developers needed to complete on schedule* (for the sprint) is calculated based on the *Time Left in Sprint* and the *Effort Remaining in Sprint*. If this number is greater than zero, then Schedule Pressure is perceived, after a short delay (controlled by *Time to Perceive Sched Pressure)*. *Schedule Pressure* in turn drives work intensity (working harder, and using overtime), which then increases the *Effective Staff*, which is the product of actual Staff by *Work Intensity*. That will increase the Sprint Work Rate and tasks will be accomplished faster.



Developers needed to complete on schedule
Time to Perceive Sched Pressure
Schedule Pressure
<Number of Team Meetings per week>
<Switch for Waterfall>
Max Work Intensity
Work Intensity
Effort Remaining in Sprint
Working Harder at End of Sprint
Normal Work Intensity
<Staff>
Effective Staff
Sprint Work Rate
<Productivity>
Sprint Backlog
Work Performed in Release
Sprint Work Being Accomplished

Figure 58 - End-of-Sprint Pressure ("Working Harder" Balancing Feedback Loop)

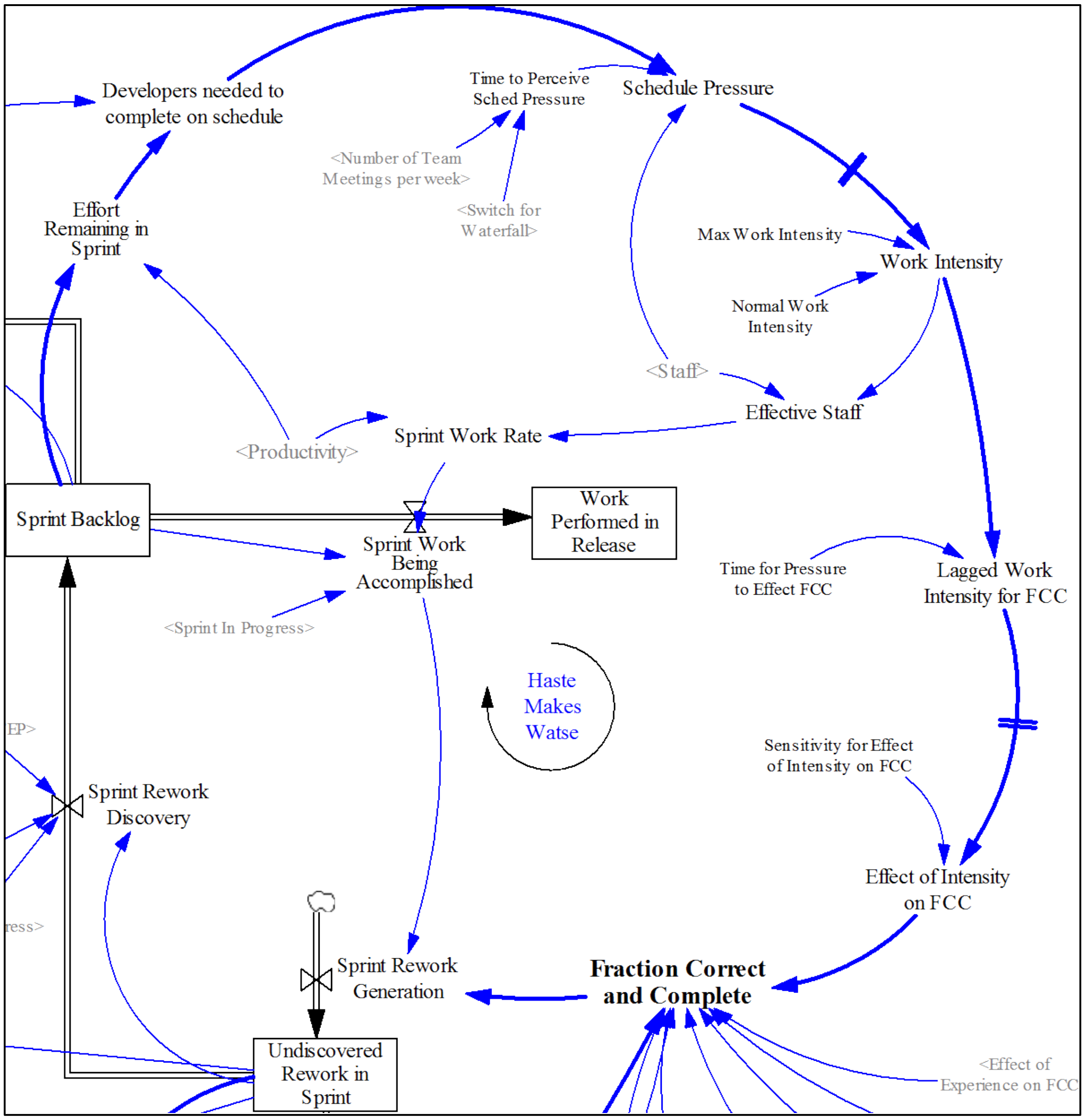
However, working at higher intensity also leads to more defect generation ("Haste Makes Waste") as shown in Figure 59, since *Work Intensity* not only speeds up the completion of correct work, but also speeds up the generation of defects:



Developers needed to complete on schedule
Schedule Pressure
Time to Perceive Sched Pressure
Effort Remaining in Sprint
<Number of Team Meetings per week>
<Switch for Waterfall>
Max Work Intensity
Work Intensity
Normal Work Intensity
<Staff>
Effective Staff
<Productivity>
Sprint Work Rate
Sprint Backlog
Work Performed in Release
Time for Pressure to Effect FCC
Sprint Work Being Accomplished
Lagged Work Intensity for FCC
<Sprint In Progress>
Burn Out
EP>
Sensitivity for Effect of Intensity on FCC
Sprint Rework Discovery
Effect of Intensity on FCC
ess>
Sprint Rework Generation
Fraction Correct and Complete
Undiscovered Rework in Sprint
<Effect of Experience on

Figure 59 - Schedule Pressure Creates More Defects ("Burn Out" Reinforcing Loop)

Moreover, working at high intensity is only sustainable for a short period of time before developers start to "Burn Out" and generate more defects than normal (i.e. becoming sloppy) –Figure 60 shows this loop in our APD model which captures the effect of increased *Work Intensity* leading to lower FCC (after a delay.)


Developers needed to complete on schedule
Time to Perceive Sched Pressure
Schedule Pressure
<Number of Team Meetings per week>
<Switch for Waterfall>
Effort Remaining in Sprint
Max Work Intensity
Work Intensity
Normal Work Intensity
<Staff>
Effective Staff
<Productivity>
Sprint Work Rate
Sprint Backlog
Work Performed in Release
Sprint Work Being Accomplished
Time for Pressure to Effect FCC
Lagged Work Intensity for FCC
<Sprint In Progress>
Haste Makes Watse
EP>
Sprint Rework Discovery
Sensitivity for Effect of Intensity on FCC
Effect of Intensity on FCC
ress>
Sprint Rework Generation
Fraction Correct and Complete
<Effect of Experience on FCC
Undiscovered Rework in Sprint
Figure 60 - Sustained High Intensity Leads to Burn Out

5.2.7 Continuous Integration

As described previously, creating a "CI Environment" means acquiring, installing, and configuring the tools and servers necessary to support CI. This may include custom scripting and development to automate certain aspects of the build process, or even modification of the software product to add harnessing or support for test automation. This can be a costly up-front effort for any project, and the project cannot enjoy the benefits of CI until this environment is available.

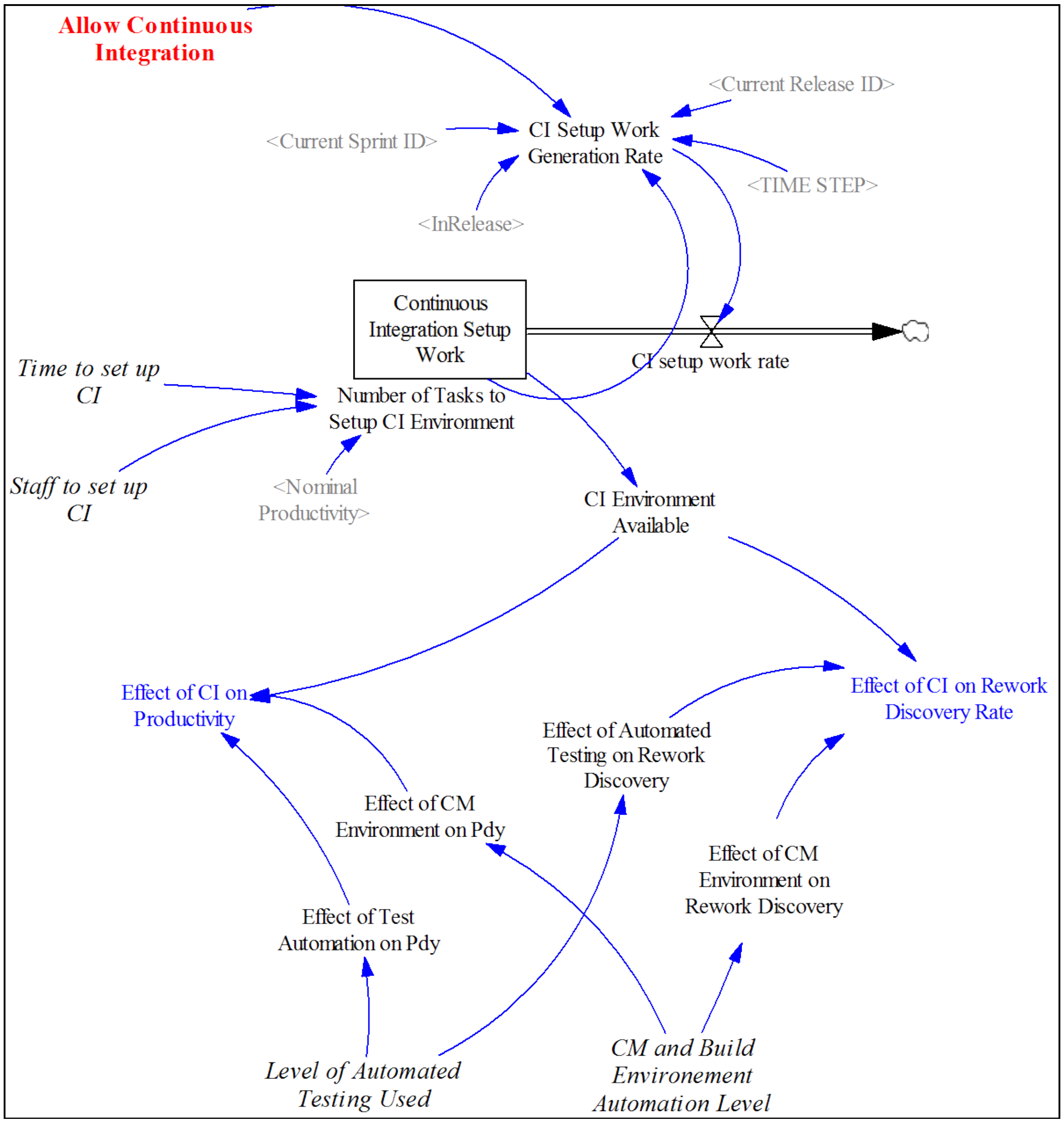

Allow Continuous Integration
<Current Release ID>
CI Setup Work Generation Rate
<Current Sprint ID>
<TIME STEP>
<InRelease>
Continuous Integration Setup Work
CI setup work rate
Time to set up CI
Number of Tasks to Setup CI Environment
Staff to set up CI
<Nominal Productivity>
CI Environment Available
Effect of CI on Productivity
Effect of CI on Rework Discovery Rate
Effect of Automated Testing on Rework Discovery
Effect of CM Environment on Pdy
Effect of CM Environment on Rework Discovery
Effect of Test Automation on Pdy
Level of Automated Testing Used
CM and Build Environement Automation Level

Figure 61 - APD Model Elements for Continuous Integration

As seen in Figure 61, our APD model captures the upfront cost of CI setup by injecting an amount of tasks into the Continuous Integration Setup Work stock. This is calculated simplistically here by using the formula:

Number of Tasks to Setup CI Environment = Nominal Productivity(Time to set up CI*Staff to set up CI)*

"*Time to Setup CI*" and "*Staff to set up CI*" both are exogenous model parameters that can be adjusted for simulation purposes. At the beginning of the project, if there is any CI setup work to do, it is the first set of tasks that are transitioned into the Release Backlog.

Also as previously described, Continuous Integration can be broken into two sub-practices: Configuration Management (CM), and Test Automation. As shown in Figure 61 CM and Build Environment Automation Level is a lever variable which can be set from 0 to 10 to indicate the level of automation (how much of the process of preparing and building the software for daily development use and especially for release is automated and repeatable.) Level of Automated Testing Used likewise represents how much test automation is in the system; this can range from 0(no automation) to 10 (fully automated unit testing as well as fully automated functional tests – the system can completely self-test as part of a nightly build.)

This allows us to 'configure' the Continuous Integration gene depending on the nature and purpose of our simulation to have varying degrees of impact on productivity and rework discovery.

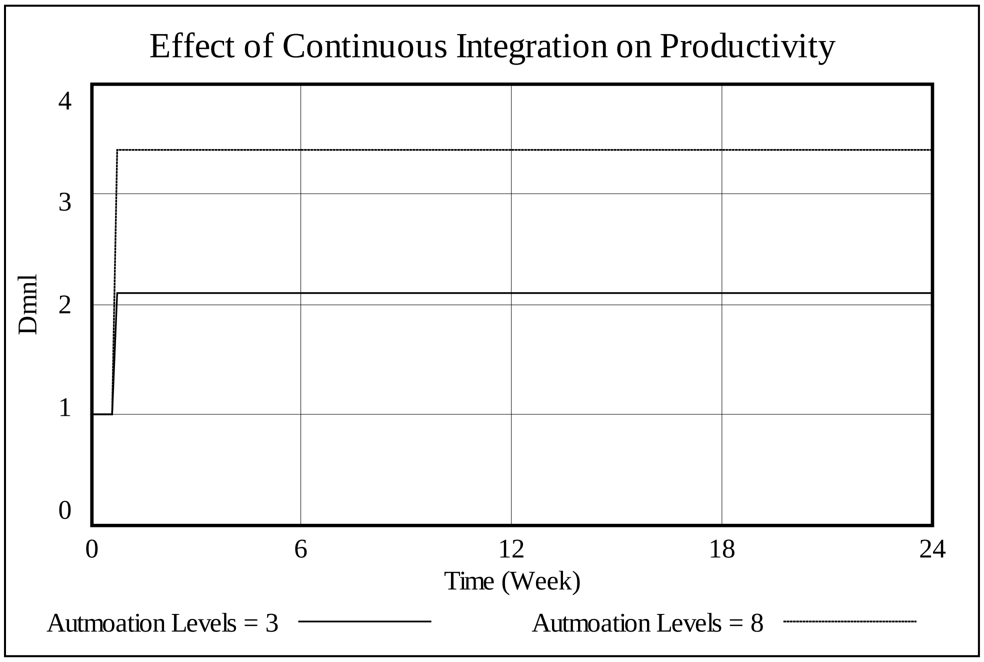
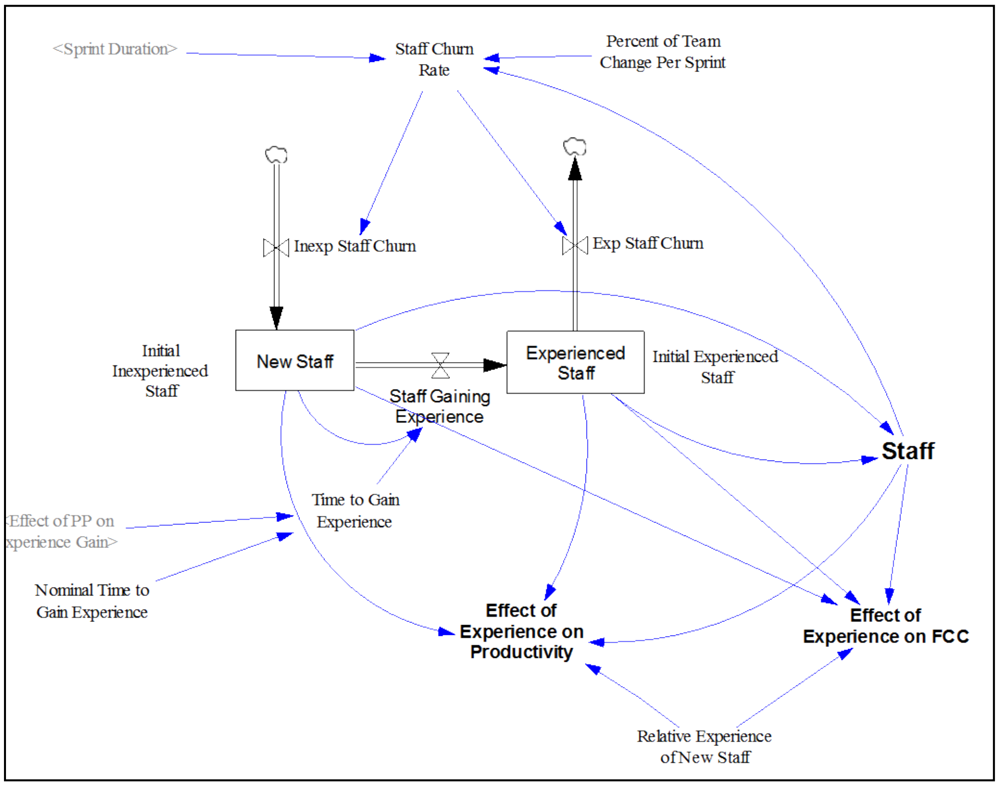


Figure 62 - Examples of Effect of Continuous Integration on Productivity

For example, Figure 62 plots two graphs for the *Effect of Continuous Integration on Productivity*. The solid line is the value of this variable after setting the *Level of Automated Testing Used* and *CM and Build Environment Automation Level* parameters both to a value of 3, while the dashed line shows its value after setting them to 8. Simply put, more automation means more productivity, as repetitive manual labor is replaced by software that automated portions of the workflow.

5.2.8 Modeling Staffing and Staff Experience

Figure 63 shows the portion of the APD model that addresses staffing and experience gain. The construct is very similar to that presented in section 4.3, describing developer experience mix and its relationship to Brooks' Law.



<Sprint Duration>
Staff Churn Rate
Percent of Team Change Per Sprint
Inexp Staff Churn
Exp Staff Churn
Initial Inexperienced Staff
New Staff
Experienced Staff
Initial Experienced Staff
Staff Gaining Experience
Staff
Time to Gain Experience
<Effect of PP on
xperience Gain>
Nominal Time to Gain Experience
Effect of Experience on Productivity
Effect of Experience on FCC
Relative Experience of New Staff

Figure 63- Staffing and Staff Experience

Figure 64 shows our staffing profile when executing this model. Here the project starts with at a pre-determined mix of *New Staff* and *Experienced Staff* (10 and 0). As time elapses, employees gain experience and become *Experienced Staff*. Note that we never reach a level of 100% of experienced employees, because our staff is changing at the rate of 5% after every sprint (*Staff Churn Rate*) to model the fact that in real organizations, developers are reassigned to different tasks and efforts regularly.

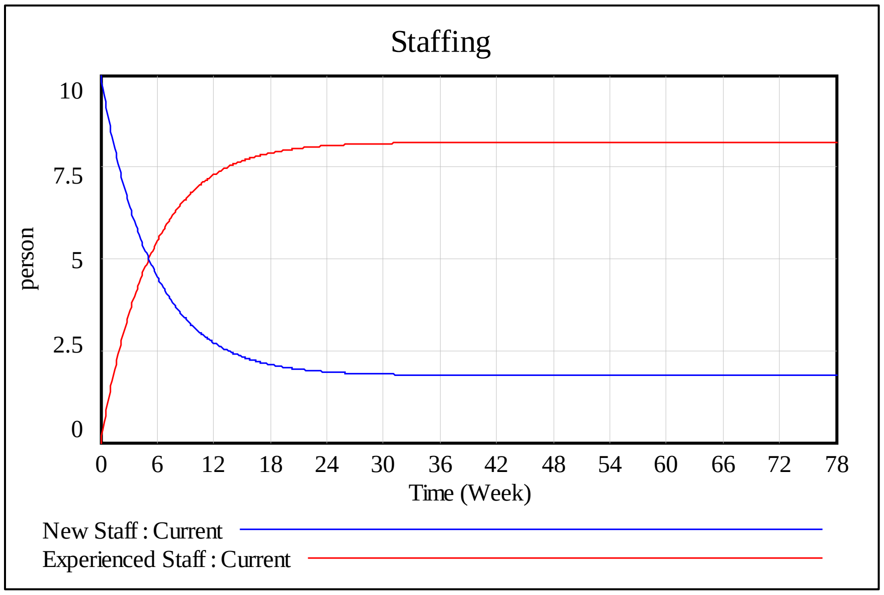


Figure 64 – Staffing Profile from APD Model

Over time, as employees gain experience, FCC and Productivity improve. Figure 65 shows how the effect of staff Experience on Productivity increases over time. The effect of experience on FCC shows a similar behavior.

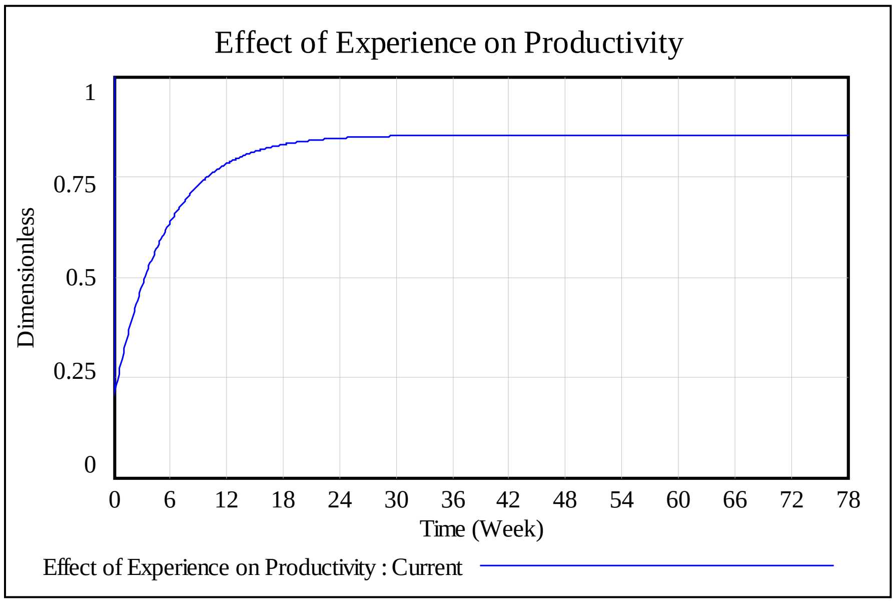


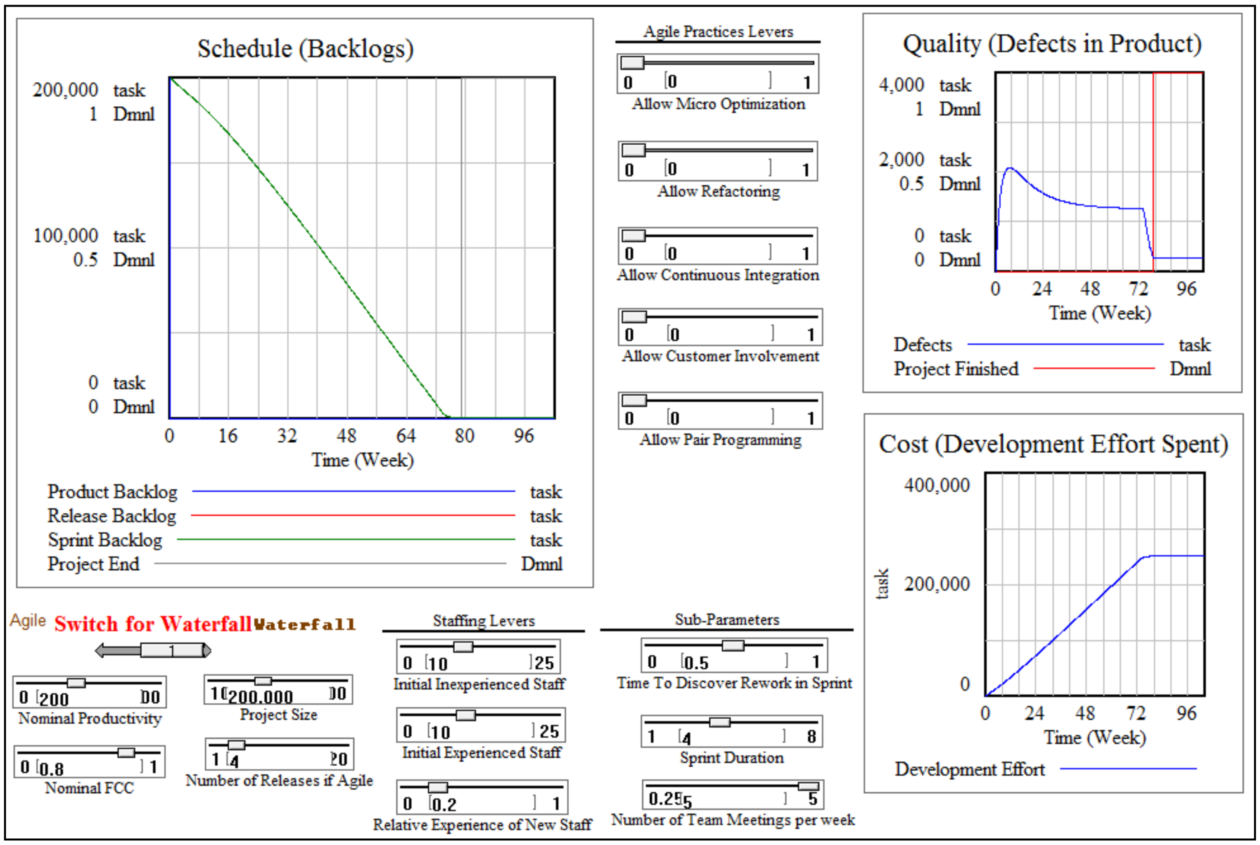
Figure 65 - Effect of Experience Gain on Productivity Increases Over Time

5.3 Summary

In this chapter, we presented a high-level overview of the APD model, including an overview of each Agile gene, and how it is modeled as a sub-system using Vensim views. The full set of model views are provided for reference in Appendix 1 – The Full Agile Project Dynamics Model. Having explained how all the pieces of the model fit together, we now move on to experimenting with our model.

6. Model Simulation Experiments

In order to perform "what if" analysis and sensitivity tests on the effects produced from the interaction of gene combinations and management policy variables, we have constructed a "Management Dashboard" that allows us to "pull the levers" on all of our project variables and observe the results (see Figure 66).



Allow Micro Optimization: 0 (0–1)
Allow Refactoring: 0 (0–1)
Allow Continuous Integration: 0 (0–1)
Allow Customer Involvement: 0 (0–1)
Allow Pair Programming: 0 (0–1)
Switch for Waterfall: 1
Nominal Productivity: 200
Project Size: 200,000
Nominal FCC: 0.8 (0–1)
Number of Releases if Agile: 4 (1–20)
Initial Inexperienced Staff: 10 (0–25)
Initial Experienced Staff: 10 (0–25)
Relative Experience of New Staff: 0.2 (0–1)
Time To Discover Rework in Sprint: 0.5 (0–1)
Sprint Duration: 4 (1–8)
Number of Team Meetings per week: 5 (0.25–5)

Figure 66 -APD Management Dashboard

The three project performance graphs that we choose to observe in our dashboard are based on the three sides of the Iron Triangle: Schedule, Cost, and Quality (see section 2.1.1). Schedule completion is determined based on the amount of time it takes for the Product, Release, and Sprint backlogs to be drained to 0. Cost is determined based on the cumulative amount of development effort spent on the project. Quality is determined based on the amount of rework (defects) that are in the product.

Before we begin our experiments, let us set up our base case parameters:

Project Size = 200,000 tasks
Initial Number of Inexperienced Staff = 10 people
Initial Number of Experienced Staff = 10 people
Normal Productivity = 200 tasks per-week, per-person

Using the 'simplistic' planning method described previously, we might project a completion time of 50 weeks (Project Size / Staff*Productivity). However, now that we recognize the existence of the rework cycle, and the effects that staff experience mix and staff churn have on the project's performance, we also set the following parameters:

Nominal Fraction Correct and Complete = 80%
Relative Experience of New Staff = 20%

We will hold these parameters constant for the rest of our experiments in order to perform a comparison of waterfall vs. agile given them same external project parameters. We will start with a baseline case of a single-pass waterfall project given these parameters, and we will proceed to "turn on" agile genes one-by-one to observe their cumulative effects of project performance. Table 6, summarizes the set of cases we will cover in the following series of experiments.

Case	Switch for Waterfall	Iterative/ incremental & Feature-driven	Micro-Optimization	Refactoring	Continuous Integration	Customer Involvement	Pair Programming
Base Case	Waterfall	OFF	OFF	OFF	OFF	OFF	OFF
Case 1	Agile	**ON**	OFF	OFF	OFF	OFF	OFF
Case 2	Agile	**ON**	**ON**	OFF	OFF	OFF	OFF
Case 3	Agile	**ON**	**ON**	**ON**	OFF	OFF	OFF
Case 4	Agile	**ON**	**ON**	**ON**	**ON**	OFF	OFF
Case 5	Agile	**ON**	**ON**	**ON**	**ON**	**ON**	OFF
Case 6	Agile	**ON**	**ON**	**ON**	**ON**	**ON**	**ON**

Table 6 - APD Model Experiements Summary

For each of our upcoming experiments, we will monitor the results in the form of three project performance variables that gauge performance along the three sides of the Iron Triangle (refer to section 2.1.1). The first measure, Schedule, gives us an idea of the project duration, i.e. how long it will take to fully complete the project. The second measure, Cost, gives us an idea of the amount of effort that will be expended to complete the project. It is measured as the cumulative amount of effort (in units of "task") expended on the project. The third and final measure is Quality. We measure quality by monitoring at the total amount of undiscovered rework in the released software across the duration of the project. By "released software" we mean the releases that have been delivered to another organization, be it integration and test teams or the final end-user.

Project Performance		
Schedule (weeks)	Cost (tasks)	Quality (H/M/L)
100	200000	M

Table 7 - Example Project Performance Triplet

An important thing to keep in mind when observing the Quality measure is that we do not simply look at the number of undiscovered rework tasks in the product, but we give it a rating of "H", "M", or "L" (High, Medium, or Low quality) based on the probability that defects are latent in the system at any given point in time. The more undiscovered rework items are in the system, the more likely that a software release will contain a number of these. Since we have not built a mechanism in the model to calculate this, for our purposes we are simply monitoring the "Quality Profile" graph of the project and making a corresponding visual judgment based on the apparent area under the curve. In Figure 67 we show a side-by-side example of quality profile graphs from two different simulations. Roughly, project A has much higher Undiscovered Rework, and over longer periods of time than project B (higher amplitude and frequency) – In this situation we score project A with "L" for low quality and project B with an "H" for high-quality.

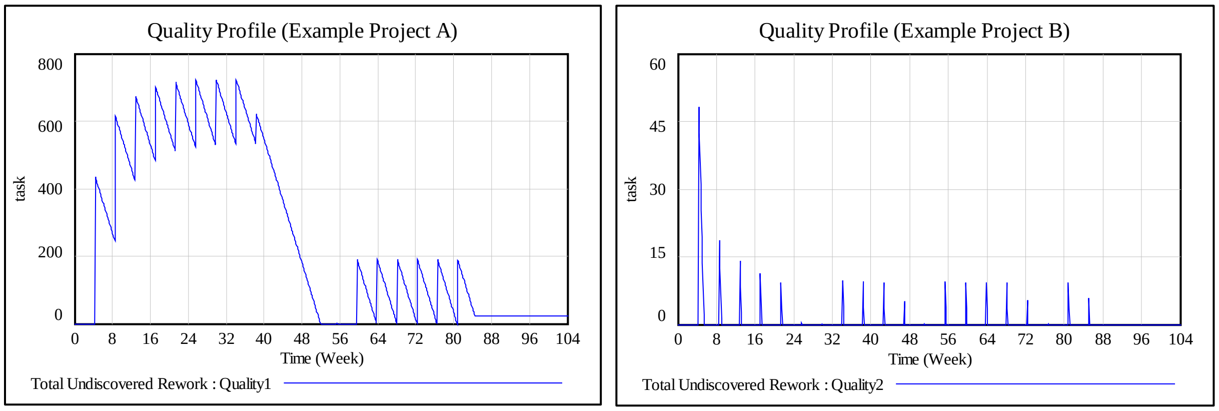


Figure 67- Examples of Quality Profile Graphs

Our APD model and set of experiments is not predicting or re-creating results from real projects, but comparing behavior of the project under different scenarios. Different exogenous parameters would produce different results, but here we are holding those exogenous parameters constant to perform a relative comparison of behavior based solely on the selection of Agile genes.

6.1 Base Case Experiment (single-pass waterfall):

As a base-case, we start by executing the model in "waterfall mode" by using our 'Switch for Waterfall". It yields the following results:

Project End Time: 79 Weeks (see Figure 68). Note that the project end time is the point at which all of the work backlogs have been depleted.

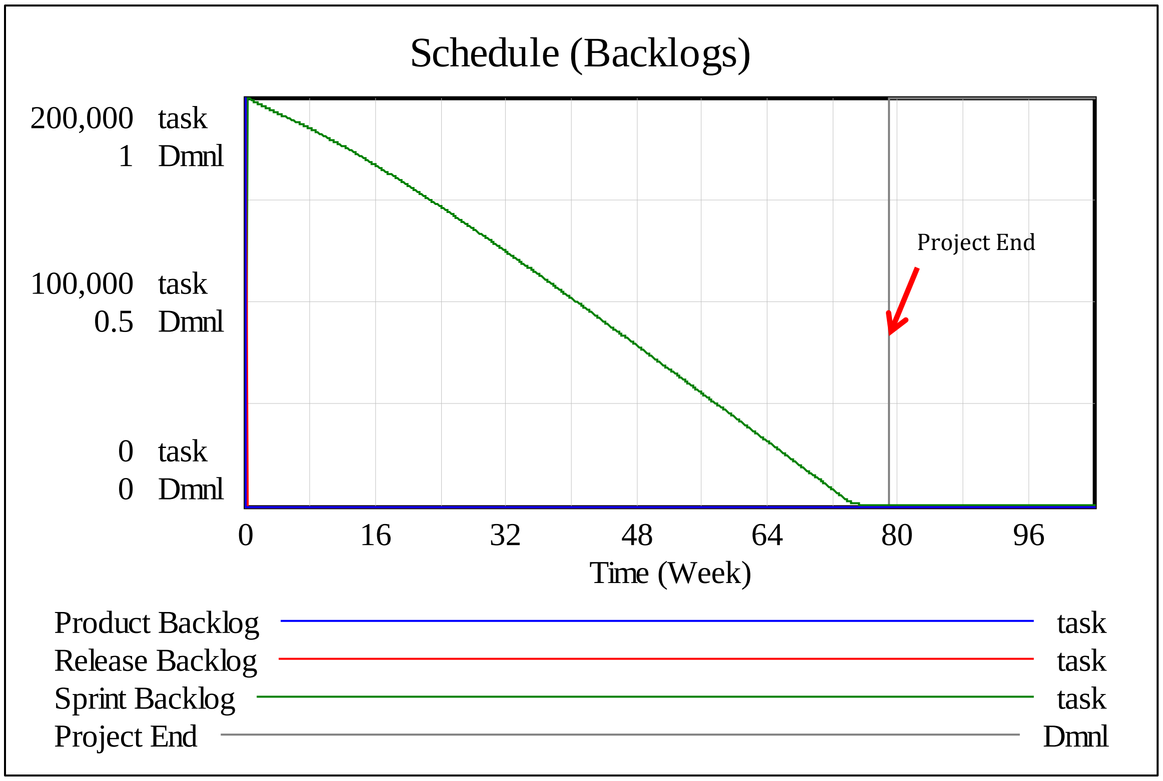


Figure 68 - APD Base Case Schedule

<u>Project Cost</u>: 250,223 tasks (see Figure 69). Note that for management, this is translatable into a dollar cost. We are currently using "task" as a fungible unit of work. This can easily be substituted with person-hours. Using this in conjunction with labor costs, we can quickly determine the dollar amount of the effort spent. For our purposes, however, "task" is an acceptable unit of cost.

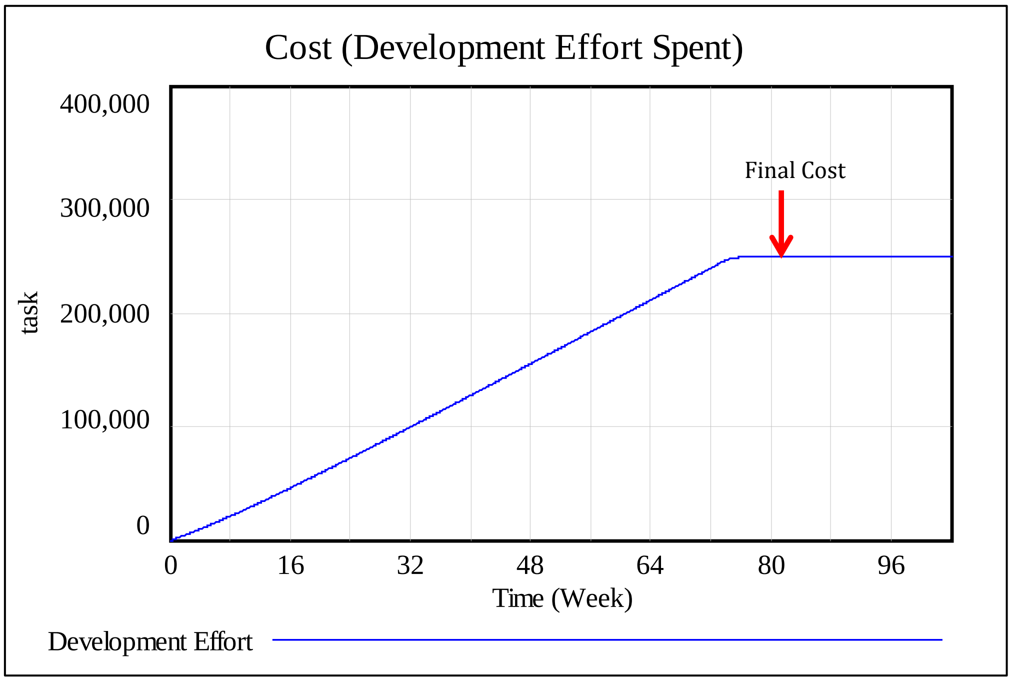


Figure 69 - APD Base Case Cost

<u>Project Quality</u>: 277 tasks (see Figure 70). Note that here we are measuring quality by the amount of undiscovered rework tasks that escape into the delivered product.

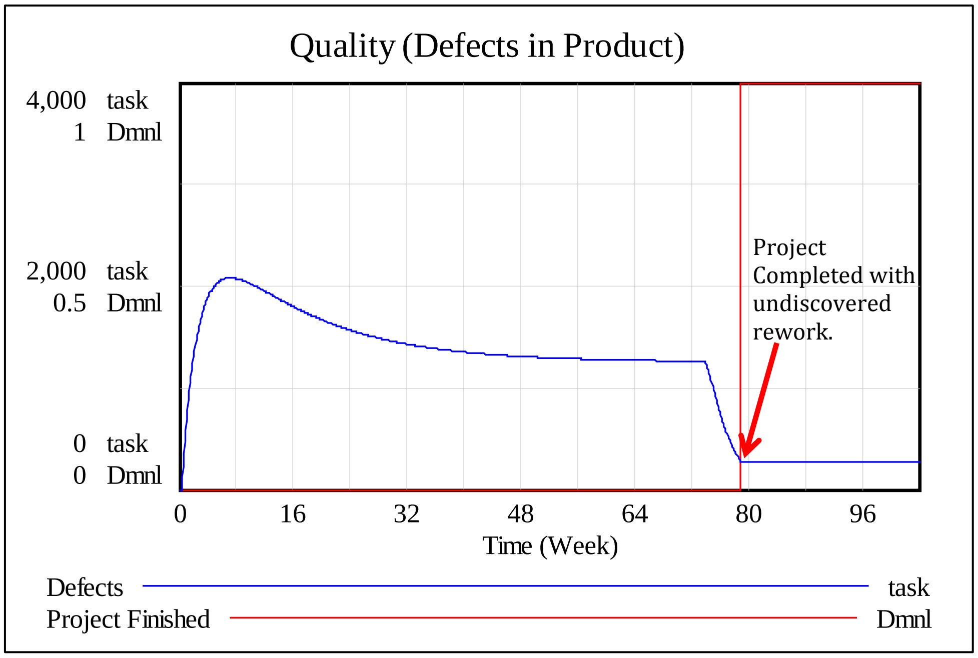


Figure 70 - APD Base Case Quality

In summary, our base case scenario produced the following project performance:

Project Performance		
Schedule (weeks)	Cost (tasks)	Quality (rework tasks)
79	250,223	L

Table 8 - Base Case Project Performance

6.2 Case 1: Fixed-schedule Feature-Driven Iterative/Incremental

In the next experiment, we 'turn off' the waterfall switch. This activates the iterative/incremental gene and the feature-driven gene, such that our project is now broken up into 4 equally sized releases that are delivered in regular intervals in feature sets. Executing the model with these settings produces the following project performance results:

	Project Performance		
Case	Schedule (weeks)	Cost (tasks)	Quality (tasks)
Base Case	79	250,223	L
Case 1	**95**	**275,111**	**M**

Table 9 - Case 1 Project Performance

Looking at these results, we find that the same project now takes 16 extra weeks, ending at week 95, and incurs extra cost (roughly 25000 more tasks). However, the product now is delivered with zero defects. Cost and Quality graphs can be observed below in Figure 71.

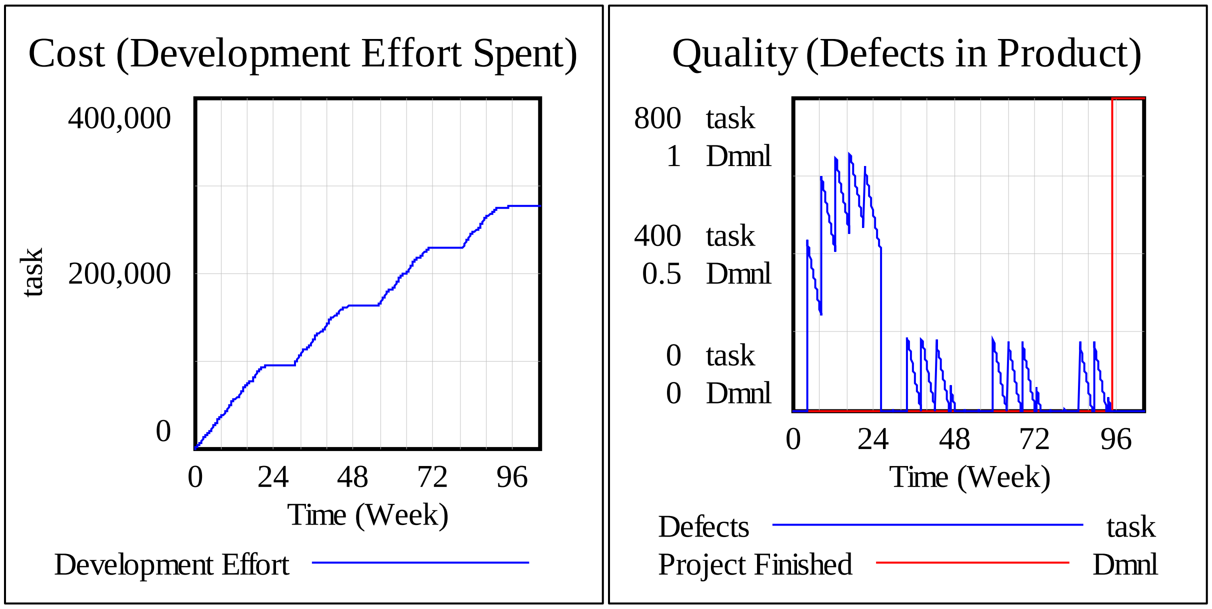


Figure 71 - Case 1 Cost and Quality

Moreover, there is a much more subtle difference generated by this case: the software is now delivered in four releases, i.e. four functional increments. These are annotated below on the Schedule graph in Figure 72. Each release is not just a random chunk of the software, but a cohesive feature set that is "potentially shippable" and has client/end-user value. Each release can be tested against system-level specifications and user expectations.

This can be of tremendous value to the recipient of the software release: they can immediately start providing feedback and acting as a "beta-tester" as of release #1 in week 24. If the recipient of this release is an integration, test, or quality assurance organization, they can get a head start on testing the software against the system specifications, performing boundary case testes, and so on. In fact, depending on the project environment, being able to several early increments of functionality may be more valuable to the customer than the added cost incurred by this case.

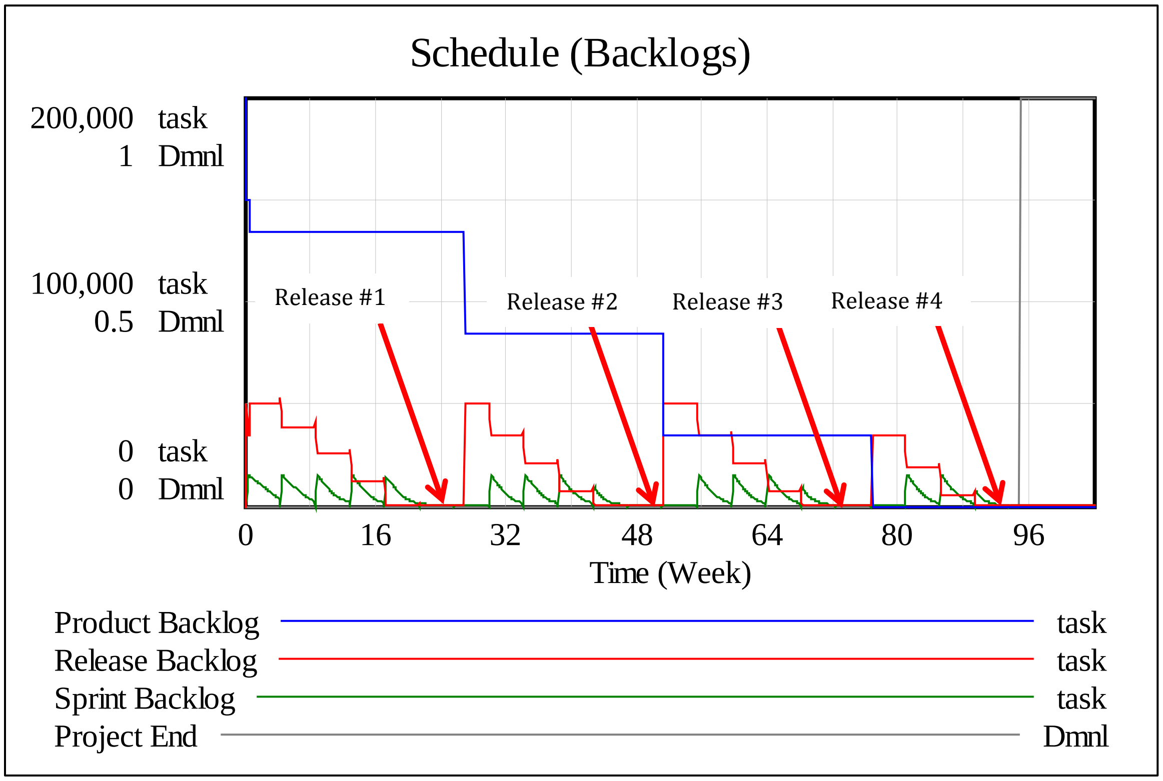


Figure 72 - Case 1 Schedule

6.3 Case 2: Introduction of Micro-Optimization.

Next, we will enable Micro-Optimization. As explained earlier (section 3.3.4), this model element simulates the management policy of empowering the development team to self-manage workload, and to perform process tweaks in between sprints, using learning from prior sprints to improve future sprint performance . We start by setting the Agile Levers for case 2. Running the model with these two genes now produces the following project performance results:

	Project Performance		
Case	Schedule (weeks)	Cost (tasks)	Quality (tasks)
Base Case	79	250,223	L
Case 1	95	275,111	M
Case 2	**82**	**244,436**	**M**

Table 10 - Case 2 Project Performance

We see improved project duration, down to 82 weeks instead of 95 in case 1. Our costs of around 244K tasks are also lower than both the base case (250K tasks) and case 1 (275K tasks).

Although this may seem counter-intuitive to management, the results here suggest that a team empowered to self-regulate their workload may curtail costs and schedule overrun as opposed to an incremental delivery model whose releases are dictated in advance. Given more time, it is almost always possible to improve the product and this could lead to the "gold-plating" effect where teams spend time over-engineering the product beyond what is necessary to achieve the desired functionality (it is almost the opposite of 'technical debt'). However, the fact that agile iterations are time-boxed and are feature-driven mitigates this issue, as teams will give priority and focus to completing specific features.

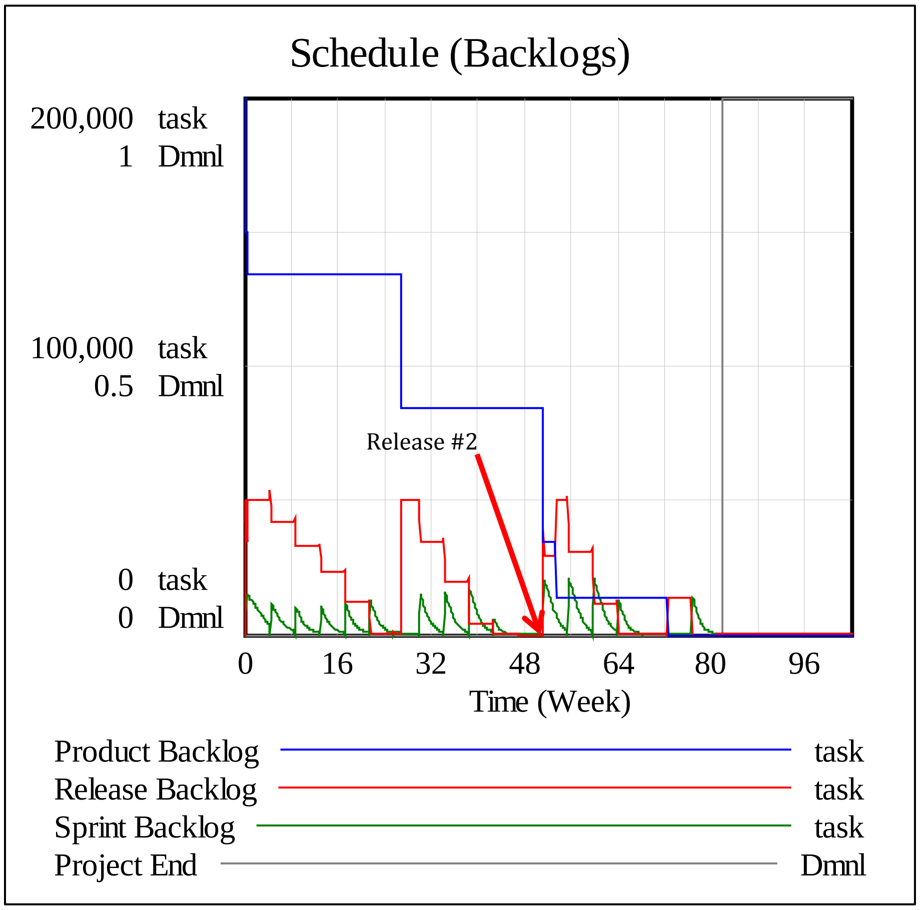


Figure 73 - Case 2 Schedule

If we examine our work backlogs in Figure 73, we see that roughly two-thirds of the functionality is delivered in the first two releases, around week 50. What has happened here, as opposed to the previous case, is that teams can gradually handle more work than was allotted to them per-sprint in the previous case, and have self-regulated the amount of work they do per-sprint. This allows them to get more sprints completed by week 50 than in the previous case.

6.4 Case 3: Introduction of Refactoring

In this case we will allow refactoring. As described earlier (5.2.3) this means that when the "technical/design debt" for the project reaches a threshold, the development team will take time to work on additional refactoring tasks to improve the quality of the software and keep it flexible and easy/cheap for expansion and addition of future features.

Executing the model thus, with three of the genes, now produces the following project performance results:

	Project Performance		
Case	Schedule (weeks)	Cost (tasks)	Quality (tasks)
Base Case	79	250,223	L
Case 1	95	275,111	M
Case 2	**82**	**244,436**	**M**
Case 3	**81.6**	**243,904**	**M**

Table 11 - Case 3 Project Performance

Intuitively, management may have thought that allowing refactoring is akin to scope creep, and thus may have thought that this would have increased either our cost or slipped the schedule. Our results, on the contrary, show that allowing refactoring resulted in a very minor (albeit negligible) improvement in schedule and in cost. To explain this, let us take a look at the graph for *Technical Debt* in case 3 vs. case 2 – see Figure 74. As can be observed in this graph, refactoring keeps our technical debt's "balance" low by refactoring. This is turn has a less detrimental effect on our *Fraction Correct and Complete* than when letting technical debt get out of hand, as shown in Figure 75.

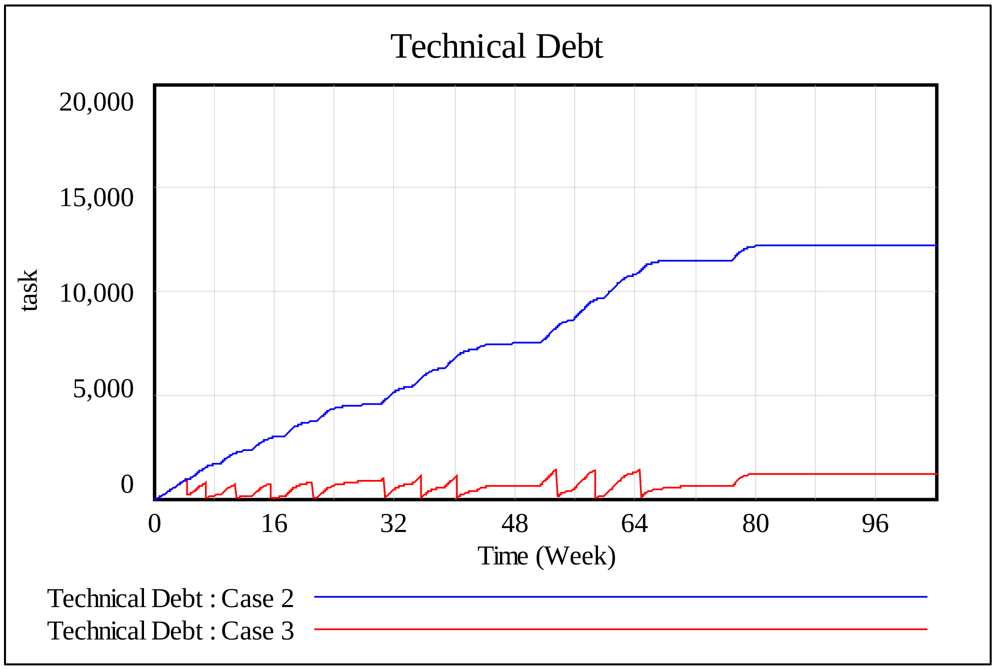


Figure 74 - Technical Debt Profile, Case 2 vs. Case 3

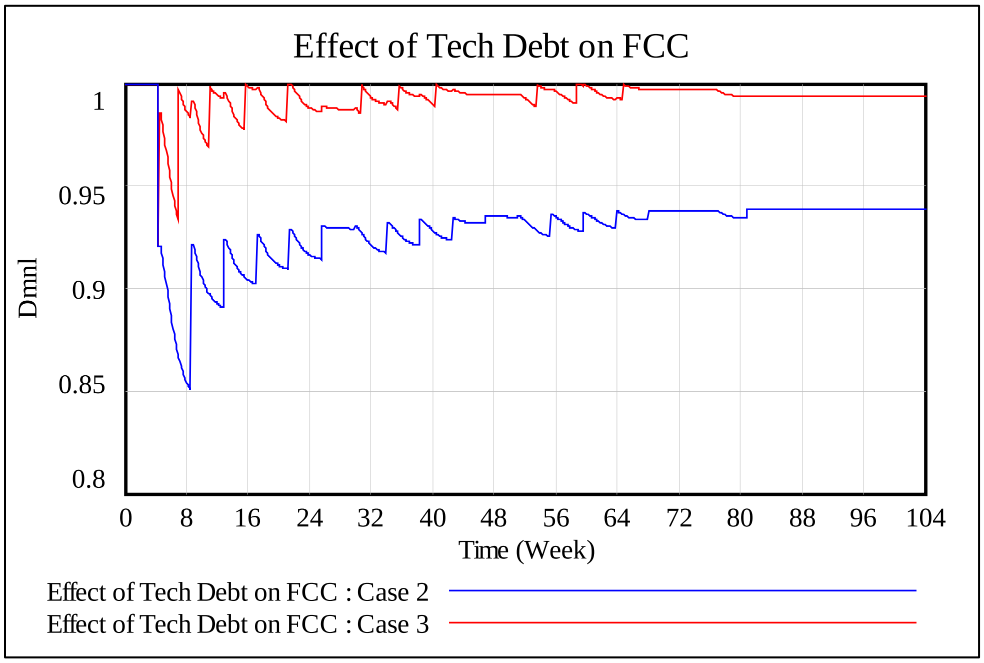


Figure 75 - Effect of Technical Debt on FCC, Case 2 vs. Case 3

Finally, there is one more subtle benefit observed in this case. Since we are now delivering the software in multiple releases, let's look at the number of undiscovered rework tasks in the software at each release. This is shown in Figure 76.

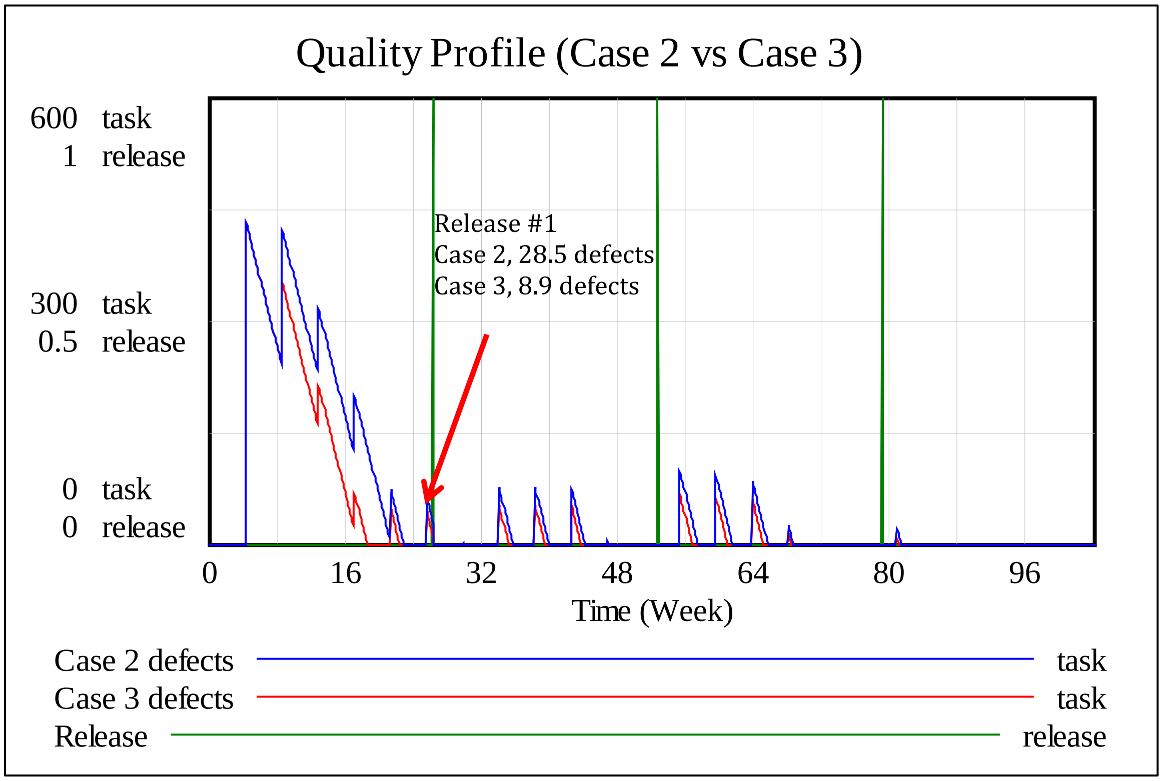


Figure 76 - Defects in Software, Case 2 vs. Case 3

Here we find that in case 2, the software was released with 28.5 defects, vs 8.9 defects in case 3 where refactoring was allowed. In other words, refactoring also allows the development team to produce better quality incremental releases. This makes sense intuitively, as it is clear that refactoring to optimize will improve the quality of the software, however the surprising part is the previous finding: that refactoring has no detrimental effect on cost or schedule. This is because of the positive effect that unpaid technical debt has on defect generation.

6.5 Case 4: Introduction of Continuous Integration

We now activate the Continuous Integration lever. This will create an initial load in tasks to be performed, representing the initial effort to set up and configure the development and delivery environment. Later, once that environment is available, it enhances productivity, and our ability to detect rework tasks (thanks to automated testing).

Executing the model with these parameters produces the following project performance results:

	Project Performance		
Case	Schedule (weeks)	Cost (tasks)	Quality (tasks)
Base Case	79	250,223	L
Case 1	95	275,111	M
Case 2	**82**	**244,436**	**M**
Case 3	**81.6**	**243,904**	**M**
Case 4	**62.8**	**255,593**	**H**

Table 12 - Case 4 Project Performance

There are no surprises here: Introducing Continuous Integration increases cost somewhat, due to the up-front investment to configure such an environment. However, the cost is recouped in schedule time. The project duration is shortened thanks to a significant speed-up in rework discovery. If the project were extended beyond 104 weeks to several years, this up-front cost becomes negligible. Another interesting observation with this gene is the quality profile as exhibited in Figure 77.

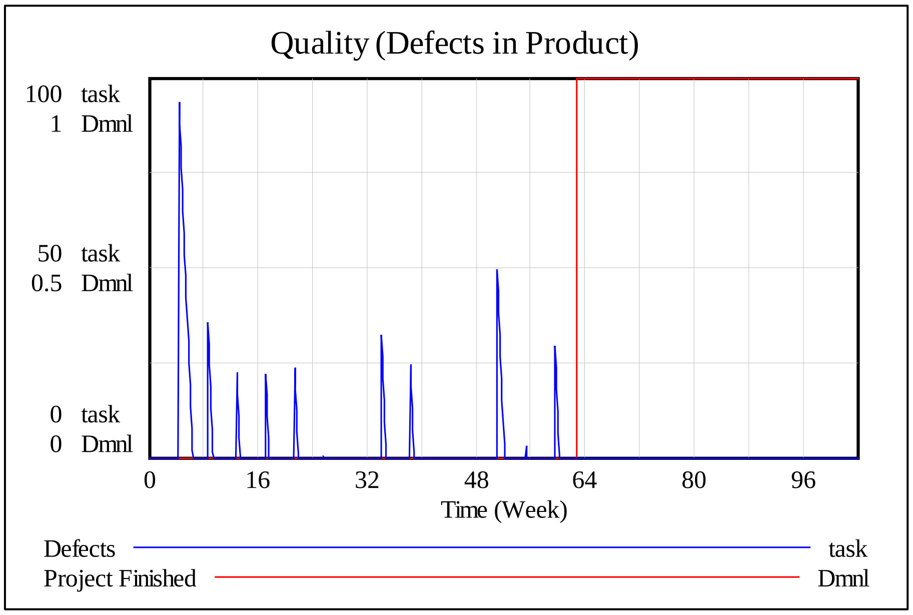


Figure 77 - Case 4 Quality

Compared to what we saw in cases 2 and 3, our quality profile shows that defects have a short life on the project, as they are quickly discovered and addressed. This has several positive effects, chiefly: the "Errors upon Errors" dynamic, described in section 4.2, is less powerful, as there are less undiscovered rework tasks dormant in the system at any given time.

6.6 Case 5: Introducing Customer Involvement

We now introduce the Customer Involvement gene, which means that there will be some requirements churn, however requirements uncertainty will be reduced as sprints progress, since the customer/user is available for feature demonstrations and immediate feedback.

Running a simulation with the above parameters yields the following results:

Case	Schedule (weeks)	Cost (tasks)	Quality (tasks)
Base Case	79	250,223	L
Case 1	95	275,111	M
Case 2	**82**	**244,436**	**M**
Case 3	**81.6**	**243,904**	**M**
Case 4	**62.8**	**255,593**	**H**
Case 5	**55.2**	**237,593**	**H**

Table 13 - Case 5 Project Performance

It seems that Customer involvement has improved both cost and schedule. The root of this is due to the effect of requirements uncertainty on the FCC.

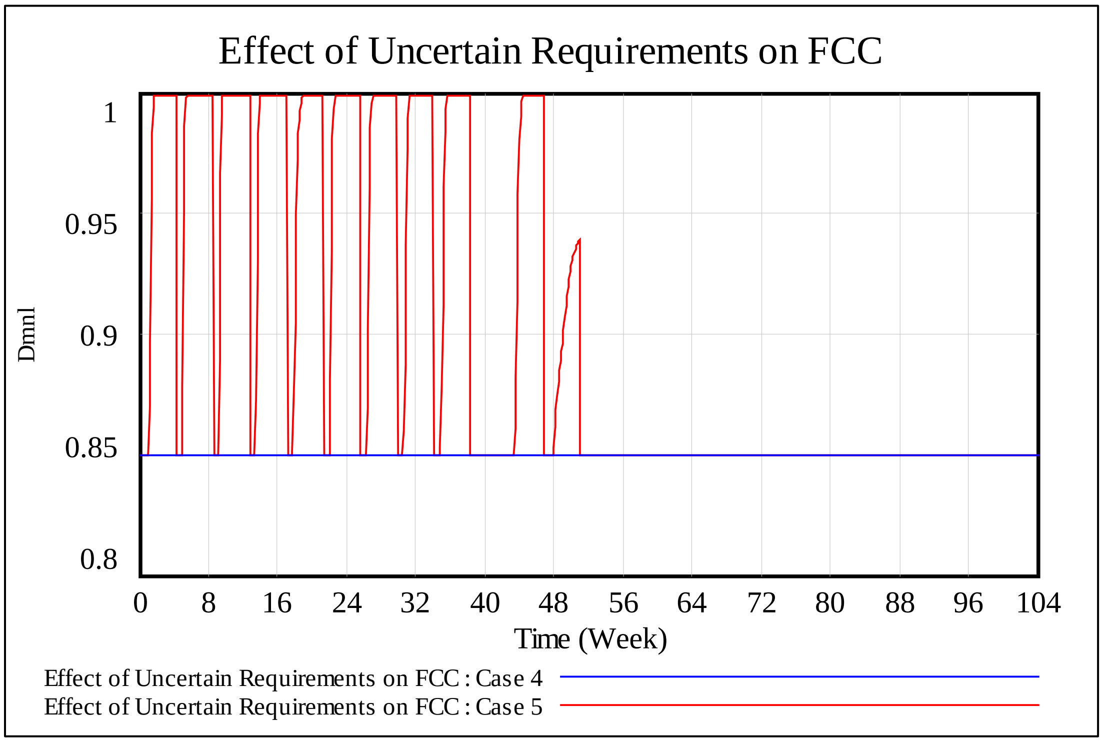


Figure 78 - Effect of Uncertain Requirements on FCC, Case 4 vs. Case 5

Figure 78 shows the graph for *Effect of Uncertain Requirements on FCC* for cases 4 and 5. In case 4, there was no customer involvement, and thus no reduction in uncertainty (effect is linear at 0.85). In case 5, as each sprint nears completion there is less uncertainty (thus higher FCC) as the development team is able to demonstrate features to customers and clarify any uncertainty on a regular informal basis.

6.7 Case 6: Introducing Pair Programming

In the final case, we look at the effect that the pair programming has on our projects' performance. In industry, pair programming is not practiced 100% of the time. Typically developers spend 20% to 50% of their time doing pair programming, thus we have set the value of *Percent Time Spent on PP* to a conservative 0.5.

Performing a simulation with these parameters produces the following project performance measures:

Case	Schedule (weeks)	Cost (tasks)	Quality (tasks)
Base Case	79	250,223	L
Case 1	95	275,111	M
Case 2	**82**	**244,436**	**M**
Case 3	**81.6**	**243,904**	**M**
Case 4	**62.8**	**255,593**	**H**
Case 5	**55.2**	**237,593**	**H**
Case 6	**52.6**	**223,749**	**H**

Table 14 - Case 6 Project Performance

We see that our project performs even better in terms of schedule compared to case 5 (52.6 weeks vs. 55.2 weeks), as well as in cost (roughly 14000 fewer tasks in development effort). This is also counter-intuitive: Pair Programming has a significant effect on productivity, as described in section 5.2.6. This is visible in our graph of development productivity in case 5 vs. case 6, seen in Figure 79.

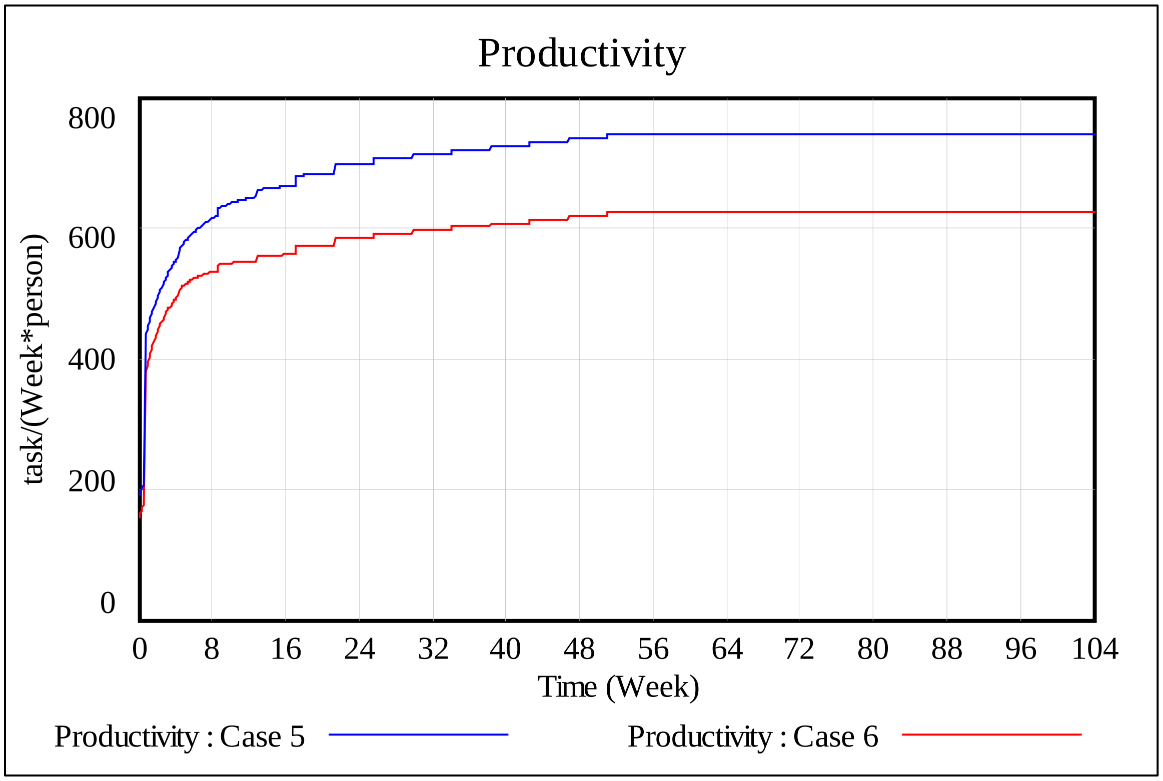


Figure 79 - Productivity Comparison, Case 5 vs. Case 6

So, why does Pair Programming improve project performance? The answer lies in the amount of rework that is generated in each case.

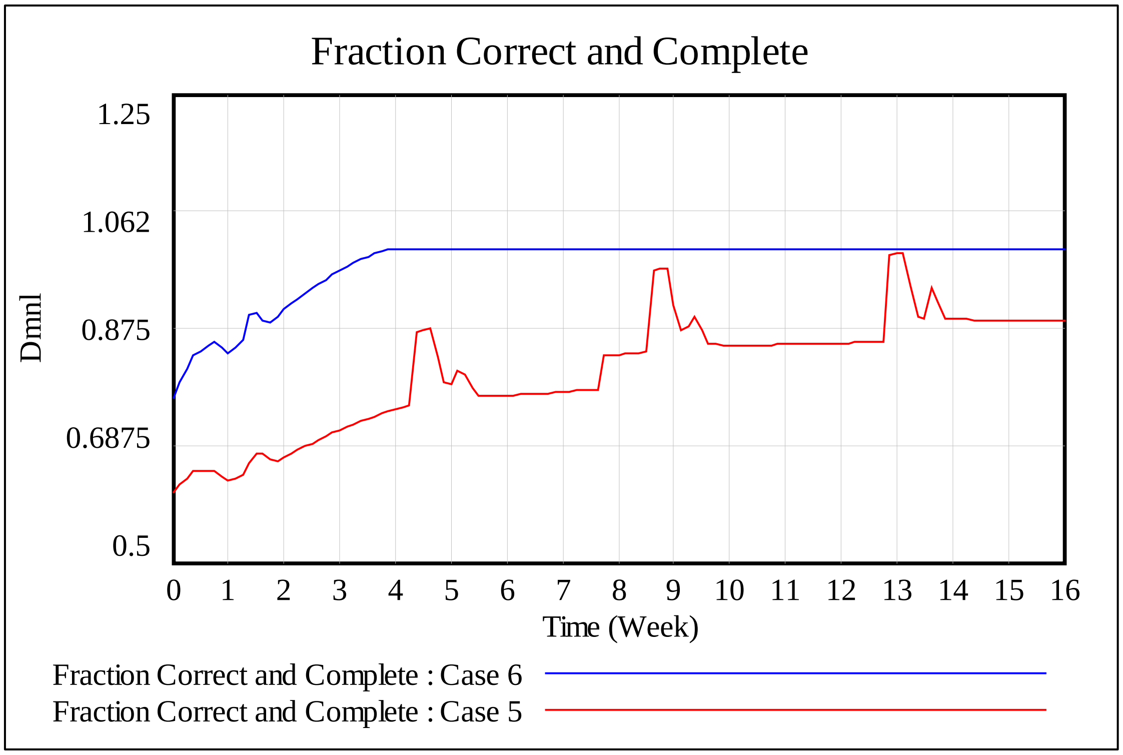


Figure 80 - FCC, Case 5 vs. Case 6

As we can see in Figure 80, our FCC is significantly better when employing Pair Programming. It is well understood that this practice produced higher quality code, as it serves as a form of real-time inspection. However, what is not immediately obvious is that better quality code begets fewer defects in subsequent development. Thus, the loss in productivity is more than made up for by the lower defect generation rates when employing this practice.

7. Conclusions and Recommendations

7.1 Interpretation of results

The six experiments performed in the previous section are but a subset of the combination and permutation of genes that can be experimented with. With six Agile gene related switches (note, that we get Feature-driven plus Interactive-incremental automatically when we switch-off Waterfall) , this means that we could experiment with 2^6, or 64 different gene combinations. Moreover, each one of the genes can have a varying effect depending on the setting of that gene's "sub-parameters. These sub-parameters are summarized below:

Micro-Optimization: *Improvement in FCC based on Sprints Completed*, *Improvement in Productivity based on Sprints Completed*, and *Improvement in Rework Discovery based on Sprints Completed* are table functions that translate the number of sprints completed into a fractional improvement in three of the rework cycle's controlling parameters.

Refactoring: *Technical Debt Accrued per Unit of Work*, *Effect of Technical Debt on FCC,* and *Refactoring Aggressiveness* represent "how much technical debt is accumulating", "what its effect on FCC is", and "how proactive is the project in refactoring their software".

Continuous Integration: Here we employ several exogenous parameters that specify a) the cost of setting up continuous integration, b) the level of automation that is employed, and c) the effect of continuous integration on FCC and Rework Discovery. The variables are: *Level of Automated Testing Used, CM and Build Environment Automation Level, Effect of Automated Testing on Rework Discovery, Effect of Automated Testing on Productivity, Effect of CM Environment on Rework Discovery,* and *Effect of CM Environment on Productivity.*

Customer Involvement: Here we have external parameters that dictate how much requirements changes (negative aspect) will result from customer involvement, as well as how their involvement will help rework discovery and reduce uncertainty. The variables are: *Effect of Customer Involvement on Requirements Churn, Elimination of Requirements Uncertainty Based on Sprint Progress,* and *Improvement in Rework Discovery Due to Customer Involvement.*

Team Dynamics: Here we capture the effect of team meetings and pair programming on the rework cycle. The sub-parameter variables are: *Effect of PP on FCC, Effect of PP on Experience Gain, Effect of PP on Productivity, Effect of Agile Teams on Pdy, Effect of Team Meetings on FCC,* and *Effect of Team Meetings on Rework Discovery*.

These sub-parameters (some exogenous variables and some lookup table functions) have been modeled using conservative values and assumptions as described in section 5.2. We have performed our experiments without changing these parameters – doing so would have exponentially increased the complexity of our experiment, way beyond the 64 high-level cases defined by the binary on/off variables employed in our simulations. Additional improvements in the model can include a second dashboard that allows experimentation with all of the sub-parameters.

Without calibrating or fine-tuning these sub-parameters, we cannot make any definitive claims about the ability of the model to reproduce the behavior of a real project, nor can we use it for predictive purposes. Each of the seven genes has both reinforcing and balancing effects on the system, via direct or indirect impact on the key parameters of the project's rework cycle, namely: Productivity, Effective Staff, Fraction Correct and Complete, Development Rework Discovery Rate. Some of the genes also have the potential of generating more tasks to be performed in either the *Product Backlog* or the *Release Backlog*.

Depending on the choice of sub-parameter values, each gene's impact can be weighted one way or the other. To clarify this point, for example, as discussed earlier: pair programming lowers productivity, but improves quality. Depending on which of these two effects is more powerful, pair programming may or may not be a beneficial practice for project performance.

Nevertheless, the development of this model has provided insight into how these Agile Genes interact to produce project behavior. This research has also generated considerable personal learning through experimentation with the model structure. In retrospect, the development of the model itself was an iterative and incremental process – the construction of each new portion of the model revealed new insights and learning about the inter-relationships between the different agile practices, and how they affect project performance.

7.2 Comparison of experiment results with personal experience

Experimentation with the APD has led to the understanding of some of the results from my personal experience with Agile. During the times when I have practiced some form of Agile development, I have never been in an environment where all seven of the Agile genes were employed.

In these environments within which I have practiced Agile, Customer Involvement was never truly practiced: in my case a System Engineer (SE) has played the role of customer proxy, and was either unwilling or unable to participate in daily scrums. Moreover, there is no guarantee that an SE could really represent the vision of the end user (In the case described in section 1.2, the SE was completely new to the Air Traffic domain).

Continuous Integration was also not truly practiced in my experience: Although we employed automatic unit-testing, very little else was automated. Functional tests were still long and laborious procedure-driven tasks. Configuration Management policy was also isolationist: in other words, pieces of functionality were developed in isolation and only integrated ("merged") near the end of the development cycle, thus no "continuous" aspect of integration to identify rework early.

Refactoring (at least, large-scale refactoring) is also a frowned-upon practice. It is considered by managers who do not understand the concept of Technical Debt to be extra non-value-added work. In one project (other than those I have mentioned previously) I found a massive amount of duplication in the C++ class hierarchy in various areas of the software. When I proceeded with a big refactoring effort, a senior engineering fellow was brought in by management to investigate my actions (as a sort of external technical auditor). Thankfully, he agreed with what I was doing and I moved on unscathed; however this is illustrative of the type of anti-refactoring sentiment in many large-scale software development communities. Likewise, I only had the opportunity to practice Pair Programming in the one case mentioned in section 1.2, otherwise this is also deemed as a productivity waste in large-scale software organizations.

This means that in most of my personal experiences with Agile, we were only able to employ elements of the Feature-Driven, Iterative-Incremental, and Micro-Optimizing genes. If we execute the APD model with the same base parameters, and only these genes activated, we get the following results:

Case	Schedule (weeks)	Cost (tasks)	Quality (tasks)
Base Case	79	250223	277
"Personal" Case	82	242213	0

Figure 81 - APD Project Performance for "Personal" Case

The results are identical to Case #2 discussed earlier during our experimentation. As we can see, this configuration saves some cost, improves quality, and yet delays the project by three weeks. This is by no means a win for agile over waterfall, or grounds to declare one superior to the other. Depending on program priorities, schedule may be the most important factor. However this explains a bit more about the promise and reality of agile practices.

7.3 Adopting Agile Practices in Large-Scale Software Engineering

Some of the "low hanging fruit", what I will call "primitive agile genes", are ones such as Team Dynamics, Feature-Driven, and Iterative-Incremental. These are relatively easy to implement or adopt, as most of these practices dictate the behavior of the software development team alone, and do not require much buy-in from other stakeholders in the product development organization (e.g. system engineering, quality assurance, management, etc.) So a development team can easily adopt these primitive genes. As is often reported, these practices are well received by developers, who will swear that they are doing "Agile development", even though from a project performance perspective, there is nothing different or agile about the cost, schedule, or quality of the software as far as the customer is concerned. Especially when practiced within a larger waterfall context, these primitive agile genes have little impact externally on the overall project behavior. In other words, Agile development can lose some of its benefits when practiced within an otherwise-waterfall program. This approach is starting to be known as "Water-Scrum-Fall" (Figure 82).

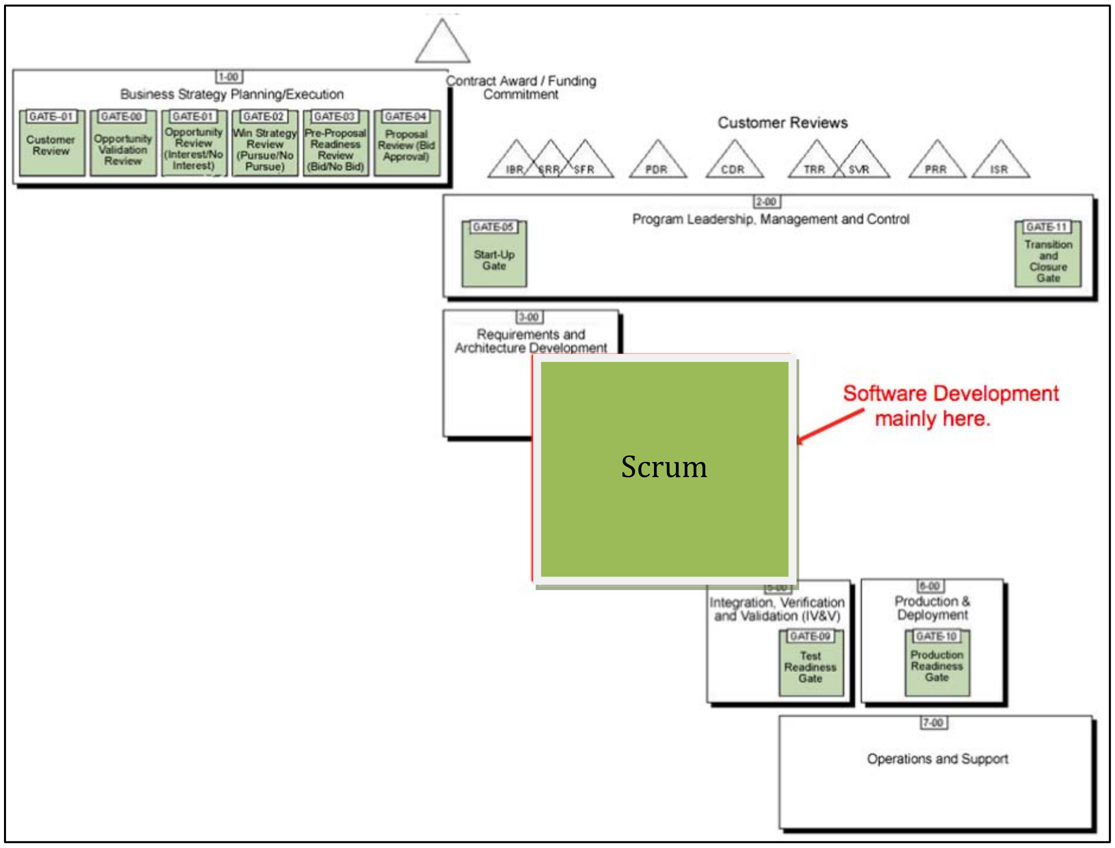

Contract Award / Funding Commitment
1-00
Business Strategy Planning/Execution
GATE-01 Customer Review
GATE-00 Opportunity Validation Review
GATE-01 Opportunity Review (Interest/No Interest)
GATE-02 Win Strategy Review (Pursue/No Pursue)
GATE-03 Pre-Proposal Readiness Review (Bid/No Bid)
GATE-04 Proposal Review (Bid Approval)
Customer Reviews
IBR
SRR
SFR
PDR
CDR
TRR
SVR
PRR
ISR
2-00
Program Leadership, Management and Control
GATE-05 Start-Up Gate
GATE-11 Transition and Closure Gate
3-00
Requirements and Architecture Development
Scrum
Software Development mainly here.
5-00
Integration, Verification and Validation (IV&V)
GATE-09 Test Readiness Gate
6-00
Production & Deployment
GATE-10 Production Readiness Gate
7-00
Operations and Support

Figure 82 - Water-Scrum-Fall

On the other hand, some of the more "advanced genes" such as Continuous Integration and Customer Involvement require much more coordination and buy-in from other stakeholders in the product development system, on a wider scale.

Continuous Integration brings with it a whole new approach and philosophy to development environments, requiring new tools, processes, and changes to existing skill-sets. It is an expensive venture that may involve retraining a team that has evolved over years of waterfall development. This is a cultural change as much as a technology or methodologies change. It means, amongst other things, that development and integration work has to be highly parallelized and integrated – This is often not the case in large-scale government software where development teams "throw work over the wall" to the integration team in assembly-line fashion.

Customer Involvement would be an even bigger cultural change, for the customer as well as the contractors: the contractor will have to be open to evolving and changing requirements as the project progresses, and the customer will have to be available (preferably present) for daily scrums and regular product demos, as well as open to incremental delivery models.

Assessment Level	Agile Gene	examples/comments
1	Team Dynamics	Daily Scrums, agile work spaces
2	Iterative-Incremental	Spiral development
3	Feature Driven	System segmented by feature
4	Refactoring	de-duplication of code, redesign
5	Micro-Optimizing	frequent process tweaks
6	Customer Involvement	on-site customer/user
7	Continuous Integration	Fully automated system tests

Table 15 - Notional Proposed Agility Assessment Model Levels

Considering the spectrum of primitive-to-advanced practices in the Agile Genome, and inspired by the concept of maturity levels, we may begin formulate an "Agility Assessment Model" (AAM) as shown in Table 15, where at each level a new Agile Gene is employed as well as all the genes of the levels below. A software development organization could be rated on a scale of 1 to 7, with 7 being the most Agile.

7.4 Follow-on work suggestions for extending the model

As discussed earlier, our APD model cannot does not replicate any specific past projects nor does it predict the performance of current projects. To do that, it must be properly calibrated. We do however claim that this model is useful for management self-directed learning and exploration. Traditional management science often favors the case study method, whereby information from real management situations are gathered and organized in a descriptive form. But the case study approach leaves that information in a descriptive form that cannot reliably cope with the dynamic complexity that is involved in a system. System dynamics modeling can organize the descriptive information, retain the richness of the real processes, build on the experiential knowledge of managers, and reveal the variety of dynamic behaviors that follow from different choices of policies (Forrester 1989).

Our model can be extended, calibrated, and enhanced to support future explorations into Agile project Dynamics. The following list summarizes suggestions as to areas of the model that can benefit from future extension and refinement:

- ***Refactoring***: The model component for this gene is built upon the concept of Technical Debt. For more accuracy in model simulations, further research is needed for understanding (a) how to quantify technical debt, and (b) how to quantify the effect of technical debt on future development in terms of its effects on the FCC and Productivity. There is a significant body of existing research on software complexity, software evolution, and software maintenance that can be leveraged for this effort.

- ***Continuous Integration***: The model component for this gene uses a simple structure and conservative parameter values to model these practices. For better simulations we need to understand and quantify the costs and benefits of test automation, build/delivery automation, and general configuration management approaches, in terms of how these affect project performance. The best way to do this may be via analysis of before-and-after project metrics for projects that have switched to employ these practices.
- ***Team Dynamics:*** We currently model this gene by considering the effects of frequent team meetings, pair programming, and schedule pressure on the development team. However the parametric values need to be calibrated using real world data. We may also seek to augment this gene by also incorporating the effects of "agile workspaces" (i.e. open collaborative spaces), "collaborative environments" (i.e. the use of collaborative software environments such as internal social media platforms).
- ***Micro- Optimizing:*** The simple structure for this gene currently improves FCC, Productivity, and Rework Discovery Time by a tiny fraction after the completion of each sprint. This mimics reports from agile practitioners that agile teams seem to gradually improve along these dimensions after many sprint cycles. Our model generates such behavior by using a "sprint counter" to generate this behavior, however this is a surrogate for the learning and knowledge gains that take place as teams become adept at practicing Agile development. Our model could be enhanced to model this as learning curve rather than a function of sprints completed. Additionally, a study of agile project data and metrics must be performed to quantify and derive plausible formulations and values for these effects.
- ***Customer Involvement***: The same is true for this gene. Although we use conservative values in our experiments, we must mine the data from actual projects to quantify the power of these effects.
- ***Staffing***: The APD model starts with predetermined staffing levels and a pre-set churn rate. The staffing component can be augmented to allow dynamics staff loading. For example, it can be enhanced to allow adjusting project staff to suit workload levels.
- ***Quality Assurance***: We currently have a simple model structure that simulates either a customer or other QA agency (perhaps an Integration or Test organization) "finding" undiscovered rework in previous releases and feeding that back into the *Release Backlog*. This can also be enhanced to cater for varying QA environments and organizational configurations.
- ***Multi-project***: The model can be extended to study a multi-project organization. In other words, software firms typically have a portfolio of projects with resources moving across projects as needed. Our model can be extended to study this situation.

7.5 Final Recommendations and Insights

This section presents the final insights and recommendations resulting from this research effort. These findings should be of value to large-scale software contractors as well as to government customers.

The first observation is that our model re-affirms some findings from existing research that uses the classic rework structure in project models, and which find rework to be at the root of project performance:

- Increasing the rate at which work is performed is not necessarily advantageous, as it also increases the rate at which errors are produced. Thus for positive results, improvement in "speed" must be paired with improvement in "quality". In other words, increase in productivity must be paired with an increase in Fraction Correct and Complete.
- *When* you find defects is as important as *how* you find them. The software industry has for a long time understood the mechanics of defect containment, and that the later a defect is found the more costly it is to correct. Our model provides insight into the dynamics behind this truism.

Agile promises to help along these two dimensions. The notion of Agile development began in the early nineties as a set of development practices and philosophies as to how to approach the task of building software. However, as we have seen, Agility is much more than just an attribute of the development methodology. It is an attribute of the whole project. As systems thinkers, we have studied the "Agile Software Development Project as a System". We find Agility to be an emergent behavior of this system, as are other "-ilities". Since the Agile Project is a complex socio-technical system, a systems-theoretic approach is required to properly understand it. Going back to our definition of a system in section 4, we said that:

1. The essential properties of a system are properties of the whole which none of its parts have. Therefore, **when a system is taken apart it loses its essential properties.**
2. No system is the sum of the behavior of its parts; it is a product of their interactions. Therefore, **when a system is taken apart, not only does it lose its properties, but so do all of the parts.**

This means that we cannot study Agile project dynamics by focusing a single aspect of the project-system in isolation from the other moving parts. For example, we cannot look at pieces of the development process without considering the developers and the way they interact (Team Dynamics: human agency is a big component of the system); we cannot isolate and study the practice of pair programming without considering the complexity of the software being developed, and the impact of previous project-related decisions (Technical Debt). And so on… We have learned, for example, that inherent Technical Debt in software systems impacts the cost and speed at which subsequent system features can be developed. This realization may lead software project managers to institutionalize the practice of Refactoring in order to keep the project Agile for future development cycles.

What we *can* do, thanks to System Dynamics, is to “surface the mental model” of experts, managers, practitioners, and customers and capture this information in an executable model, the APD model, that can be used for experimentation and as a tool for learning. Through this research and the experience modeling the project-system with System Dynamics, my own mental model and understanding of software engineering ecosystem have benefited greatly. System Dynamics takes our assumptions about the system and exposes them by making them explicit in the model structure and feedback. Through interviews, literature review, and access to past project performance data, such a model can eventually be fine-tuned to serve both as a predictive model to aid in planning and “what-if” scenario exploration by decision makers.

We find that, for a project to be Agile, not only must it employ Agile development methods, but must also fit within an Agile product development system: The development organization must be willing to practice refactoring, or lose the benefits of Agile. The software itself must be Agile, lending itself to rapid incremental deliveries and therefore must be architected accordingly in feature sets.

The customer or recipient of the software product must also be agile, willing and open to participate in the development process, and accept incremental releases that deliver these feature sets. Although our APD model does not address this, there are clearly dependencies between software features, and some feature combinations may be less useful to the customer than others. The project’s stakeholders, on both developer and customer sides, must be able to work together to plan and prioritize incremental deliveries; In the commercial world, this is less of an issue, as a commercial software product developer need not necessarily coordinate project specifics with an external entity when prioritizing and planning software releases.

Moreover, a follow-on insight (as hinted in a previous section) is that for projects to be Agile, they cannot simply adopt so-called development techniques at random (or by selection of ‘lowest-hanging fruit’) and hope to derive positive project performance results. We have learned that each practice, combined with project policies, have both positive and negative effects on the project. The trick is to configure the project-system in a fashion that maximizes the up-sides while minimizing the downsides of these practices. In keeping with our metaphor of the “Agile Genome,” we might say that “Genetic Project Engineering” is required to maximize the potential of Agile practices. The selection of management policies and combination of agile practices by software development organizations need to be balanced to optimize the system. Agile genes need to be paired to counter-balance negative effects and maximize positive effects. Some examples of this that we have explored include: The inefficiencies of the Feature Driven gene are counter-balanced by the Refactoring gene. The loss in productivity from Pair Programming is counter-balanced by the Continuous Integration gene.

8. Works Cited

Ackoff, R.L., 1971. Towards a System of Systems Concepts. *Management Science*, 17(11).

Allen, T.J. & Henn, G., 2006. *The Organization and Architecture of Innovation: Managing the Flow of Technology*,

Beck, K., 2001. Aim, Fire.

Beck, K., 1999. *Extreme Programming Explained*,

Berwin, B., 2010. How a lawn mowing contract is changing Defense acquisition. *NEXTGov*, pp.1-2. Available at: http://www.nextgov.com/nextgov/ng_20100916_6260.php.

Boehm, B.W., 1987. A Spiral Model of Software Development and Enhancement. , (1).

Brooks, F.P., 1987. No Silver Bullet. , (April), pp.1-14.

Brooks, F.P., 1975. *The Mythical Man Month: Essays on Software Engineering*,

Chatfield, B.C. & Johnson, T., 2007. A short course in project management. , pp.1-6. Available at: http://office.microsoft.com/en-us/project-help/a-short-course-in-project-management-HA010235482.aspx.

Cockburn, A., 2004. The Crystal Methods , or How to make a methodology fit.

Cooper, K.G., 1980. Naval ship production: a claim settled and a framework built. *Interfaces*, (December 1980), pp.20-36.

Cusumano, M.A. & Smith, S., 1995. Beyond the Waterfall : Software Development at Microsoft.

Davis, A., 1993. Software-Lemmingineering.pdf.

Deemer, P. et al., 2009. The Scrum Primer. , pp.1-22.

Fine, C.H., 1996. Industry Clockspeed and Competency Chain Design : An Introductory Essay.

Forrester, J.W., 1989. The Beginning of System Dynamics. In *International Meeting of the System Dynamics Society*. pp. 1-16.

Gershon, M., 2011. Double LEAN Six Sigma – A Structure for Applying Lean Six Sigma. , 12(6), pp.26-31.

Goyal, S., 2007. Major Seminar On Feature Driven Development Agile Techniques for Project Management Software Engineering.

Highsmith, J., 2002a. *Agile Software Development Ecosystems*,

Highsmith, J., 2002b. What Is Agile Software Development? *CrossTalk. The Journal of Defense Software*, (October). Available at: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:What+Is+Agile+Software+Development+?#0 [Accessed April 18, 2012].

Krigsman, M., 2012. Worldwide cost of IT failure (revisited): $ 3 trillion. *ZDNet*, pp.1-9. Available at: http://www.zdnet.com/blog/projectfailures/worldwide-cost-of-it-failure-revisited-3-trillion/15424.

Larman, C. & Basili, V., 2003. Iterative and Incremental Development : A Brief History. , (June), pp.47-56.

Lyneis, J.M., Cooper, K.G. & Els, S. a., 2001. Strategic management of complex projects: a case study using system dynamics. *System Dynamics Review*, 17(3), pp.237-260. Available at: http://doi.wiley.com/10.1002/sdr.213 [Accessed March 9, 2012].

Martin, L.A., 1997. The First Step.

Maximilien, E.M. & Williams, L., 2003. Assessing Test-Driven Development at IBM.

Mcfarland, I., 2006. Agile Practices on Real-World Projects. In *Google Java Users Group Agile 101*.

Raynus, J., 1998. *Software Process Improvement With CMM*,

SEI, 2002. Upgrading from SW-CMM ® to CMMI ®.

Sterman, J.D., 1992. System Dynamics Modeling for Project Management. , 1951.

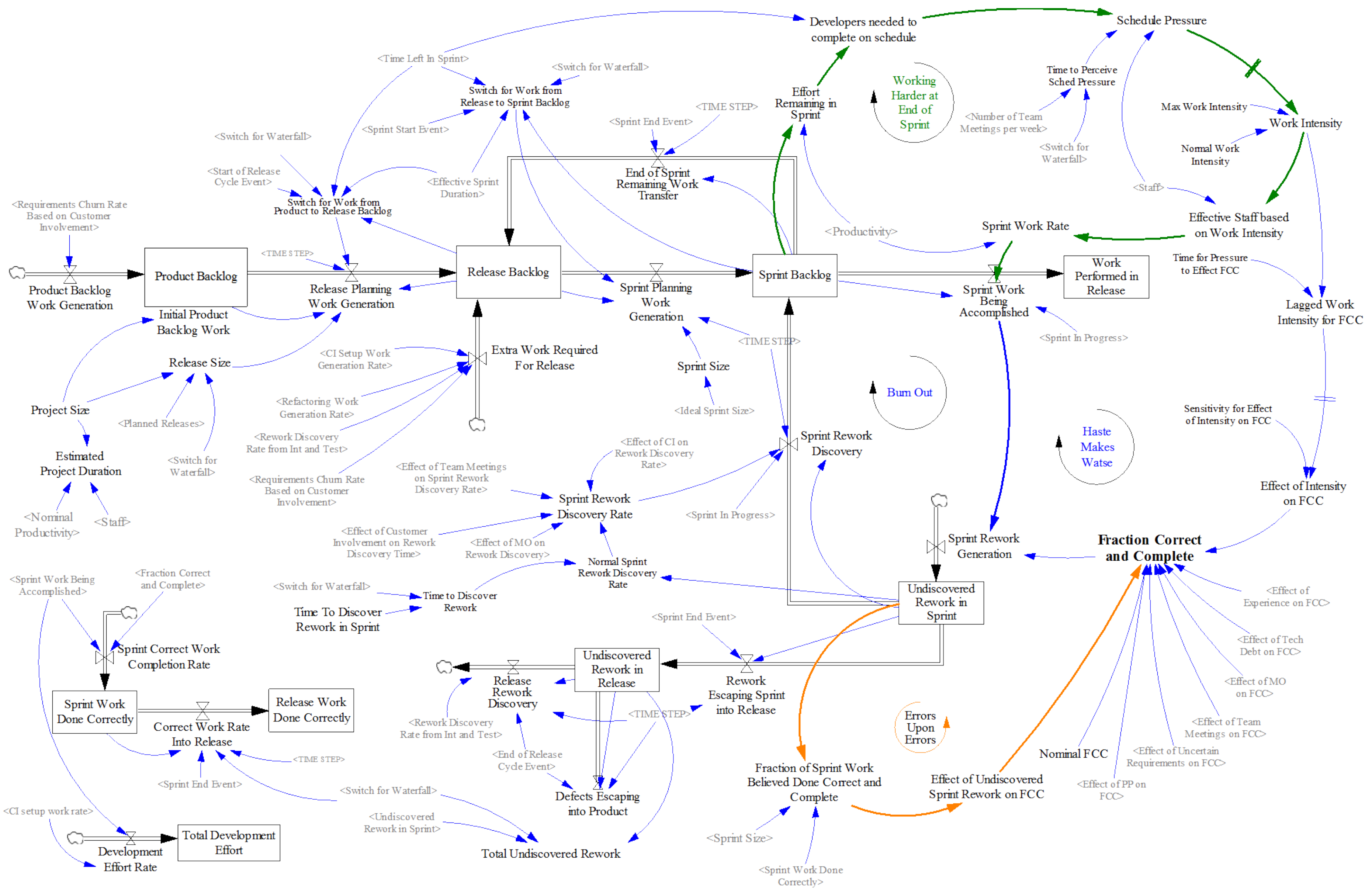
Takeuchi, H. & Nonaka, I., 1984. The new new product development game. , pp.137-147.

de Weck, O. & Lyneis, J., 2011. *Successfully Designing and Managing Complex Projects* First Edit., MIT Press.

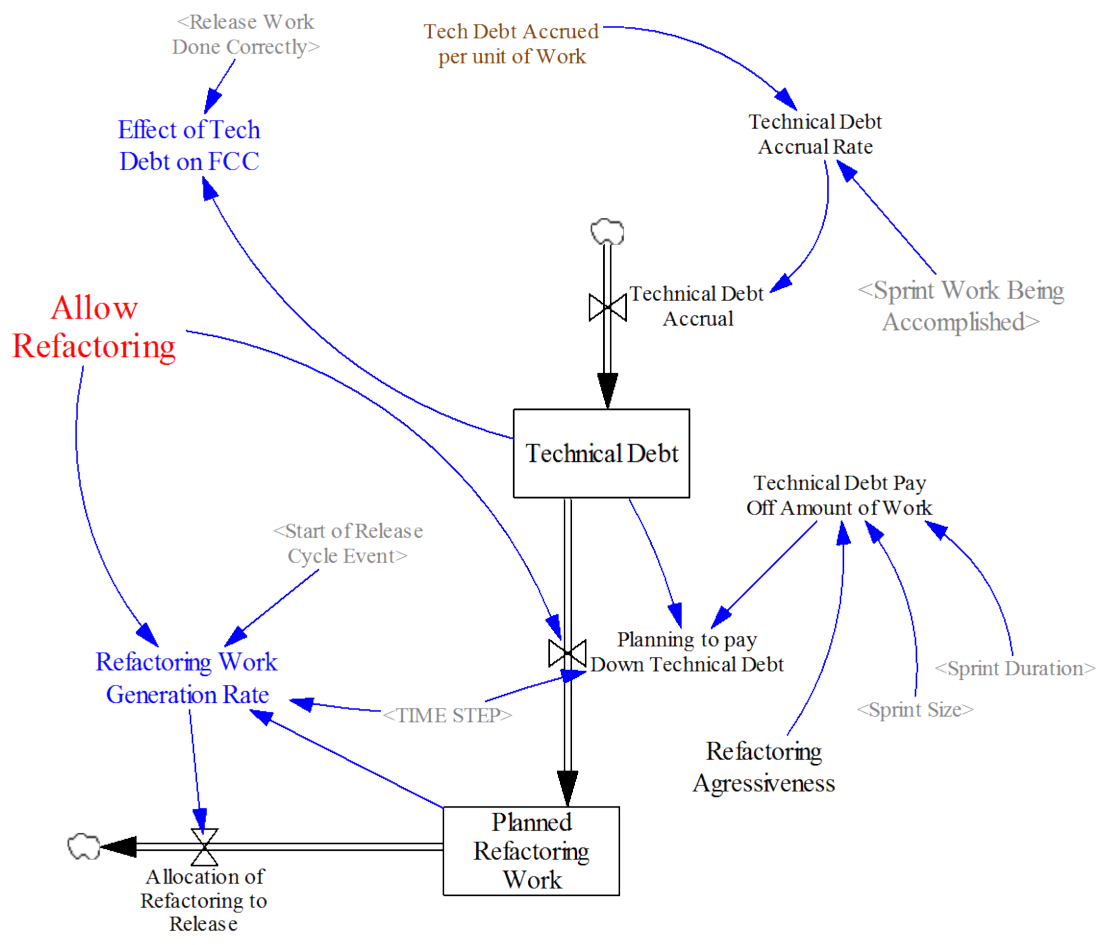
West, D. et al., 2011. Water-Scrum-Fall Is The Reality Of Agile For Most Organizations Today.

Appendix 1 – The Full Agile Project Dynamics Model

8.1 View 1: The Software Development Cycle

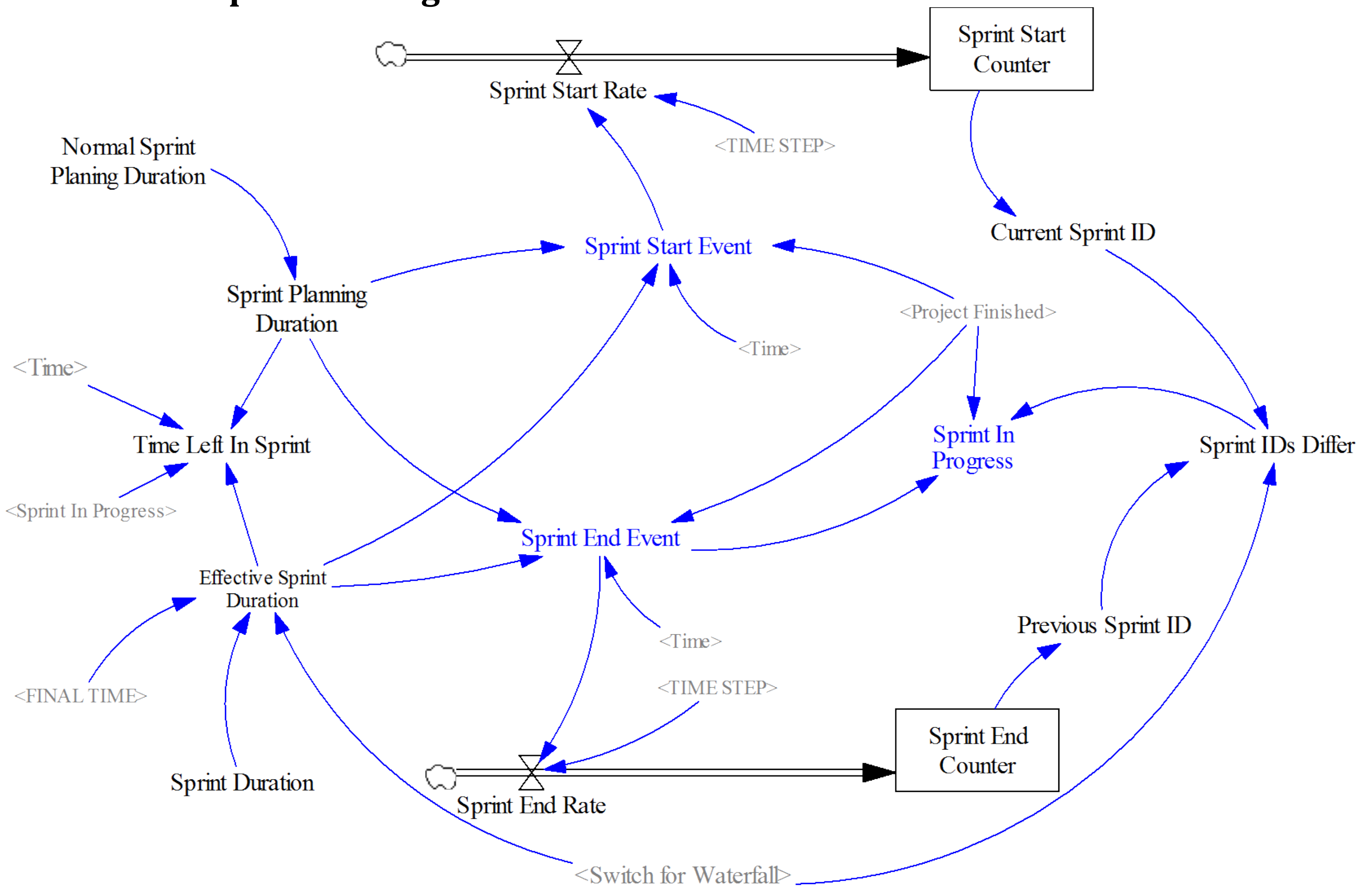

Developers needed to complete on schedule
Schedule Pressure
<Time Left In Sprint>
<Switch for Waterfall>
Switch for Work from Release to Sprint Backlog
<TIME STEP>
Effort Remaining in Sprint
Working Harder at End of Sprint
Time to Perceive Sched Pressure
<Number of Team Meetings per week>
Max Work Intensity
Work Intensity
Normal Work Intensity
<Switch for Waterfall>
<Sprint Start Event>
<Sprint End Event>
<Switch for Waterfall>
<Start of Release Cycle Event>
<Effective Sprint Duration>
End of Sprint Remaining Work Transfer
<Staff>
Effective Staff based on Work Intensity
<Requirements Churn Rate Based on Customer Involvement>
Switch for Work from Product to Release Backlog
<Productivity>
Sprint Work Rate
<TIME STEP>
Product Backlog
Release Backlog
Sprint Backlog
Work Performed in Release
Time for Pressure to Effect FCC
Product Backlog Work Generation
Release Planning Work Generation
Sprint Planning Work Generation
Sprint Work Being Accomplished
Lagged Work Intensity for FCC
Initial Product Backlog Work
<CI Setup Work Generation Rate>
Extra Work Required For Release
<TIME STEP>
<Sprint In Progress>
Release Size
Sprint Size
Burn Out
<Refactoring Work Generation Rate>
<Ideal Sprint Size>
Project Size
<Planned Releases>
<Rework Discovery Rate from Int and Test>
Haste Makes Watse
Sensitivity for Effect of Intensity on FCC
<Effect of CI on Rework Discovery Rate>
Sprint Rework Discovery
Estimated Project Duration
<Switch for Waterfall>
<Requirements Churn Rate Based on Customer Involvement>
<Effect of Team Meetings on Sprint Rework Discovery Rate>
Sprint Rework Discovery Rate
Effect of Intensity on FCC
<Nominal Productivity>
<Staff>
<Sprint In Progress>
<Effect of Customer Involvement on Rework Discovery Time>
<Effect of MO on Rework Discovery>
Sprint Rework Generation
Fraction Correct and Complete
Normal Sprint Rework Discovery Rate
<Sprint Work Being Accomplished>
<Fraction Correct and Complete>
<Switch for Waterfall>
Time to Discover Rework
Undiscovered Rework in Sprint
<Effect of Experience on FCC>
Time To Discover Rework in Sprint
<Sprint End Event>
Sprint Correct Work Completion Rate
Undiscovered Rework in Release
<Effect of Tech Debt on FCC>
Release Rework Discovery
Rework Escaping Sprint into Release
Sprint Work Done Correctly
Release Work Done Correctly
<Effect of MO on FCC>
Correct Work Rate Into Release
<TIME STEP>
Errors Upon Errors
<Reowrk Discovery Rate from Int and Test>
<Effect of Team Meetings on FCC>
<TIME STEP>
<End of Release Cycle Event>
Nominal FCC
<Effect of Uncertain Requirements on FCC>
<Sprint End Event>
<Switch for Waterfall>
Fraction of Sprint Work Believed Done Correct and Complete
Effect of Undiscovered Sprint Rework on FCC
Defects Escaping into Product
<Effect of PP on FCC>
<CI setup work rate>
<Undiscovered Rework in Sprint>
Total Development Effort
Development Effort Rate
Total Undiscovered Rework
<Sprint Size>
<Sprint Work Done Correctly>

8.2 View 2: Refactoring

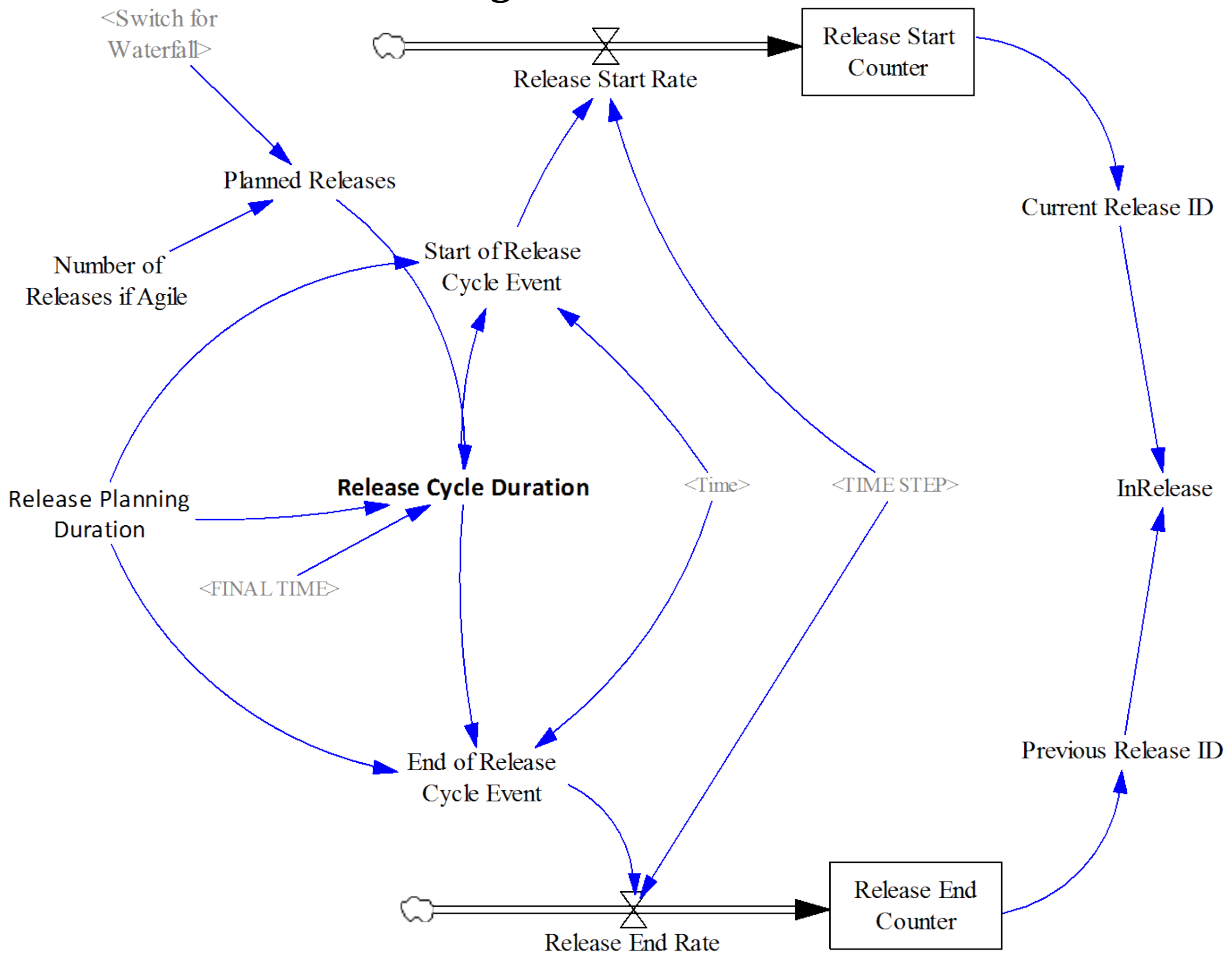


<Release Work Done Correctly>
Tech Debt Accrued per unit of Work
Effect of Tech Debt on FCC
Technical Debt Accrual Rate
Technical Debt Accrual
<Sprint Work Being Accomplished>
Allow Refactoring
Technical Debt
Technical Debt Pay Off Amount of Work
<Start of Release Cycle Event>
Planning to pay Down Technical Debt
Refactoring Work Generation Rate
<TIME STEP>
<Sprint Duration>
<Sprint Size>
Refactoring Agressiveness
Planned Refactoring Work
Allocation of Refactoring to Release

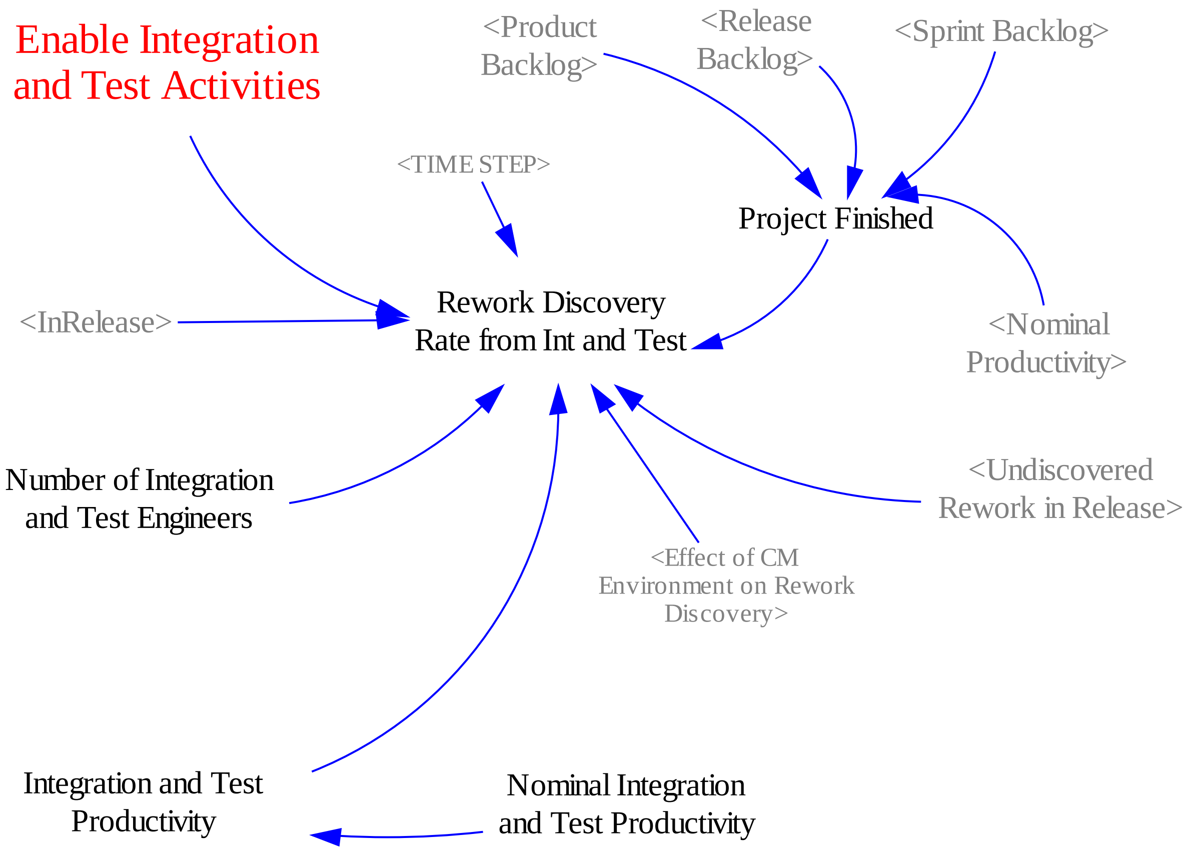
8.3 View 3: Sprint Timing



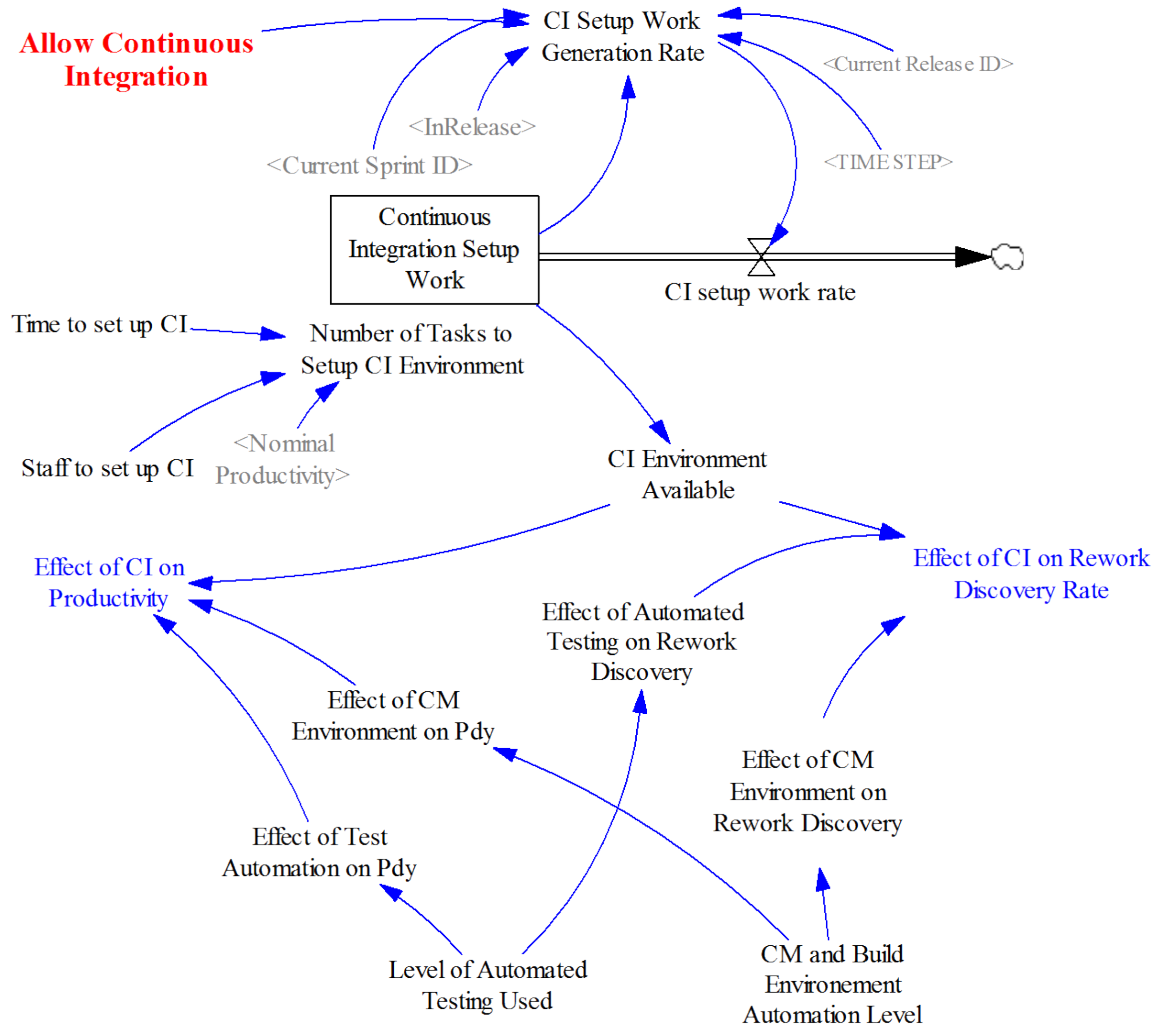
8.4 View 4: Release Timing



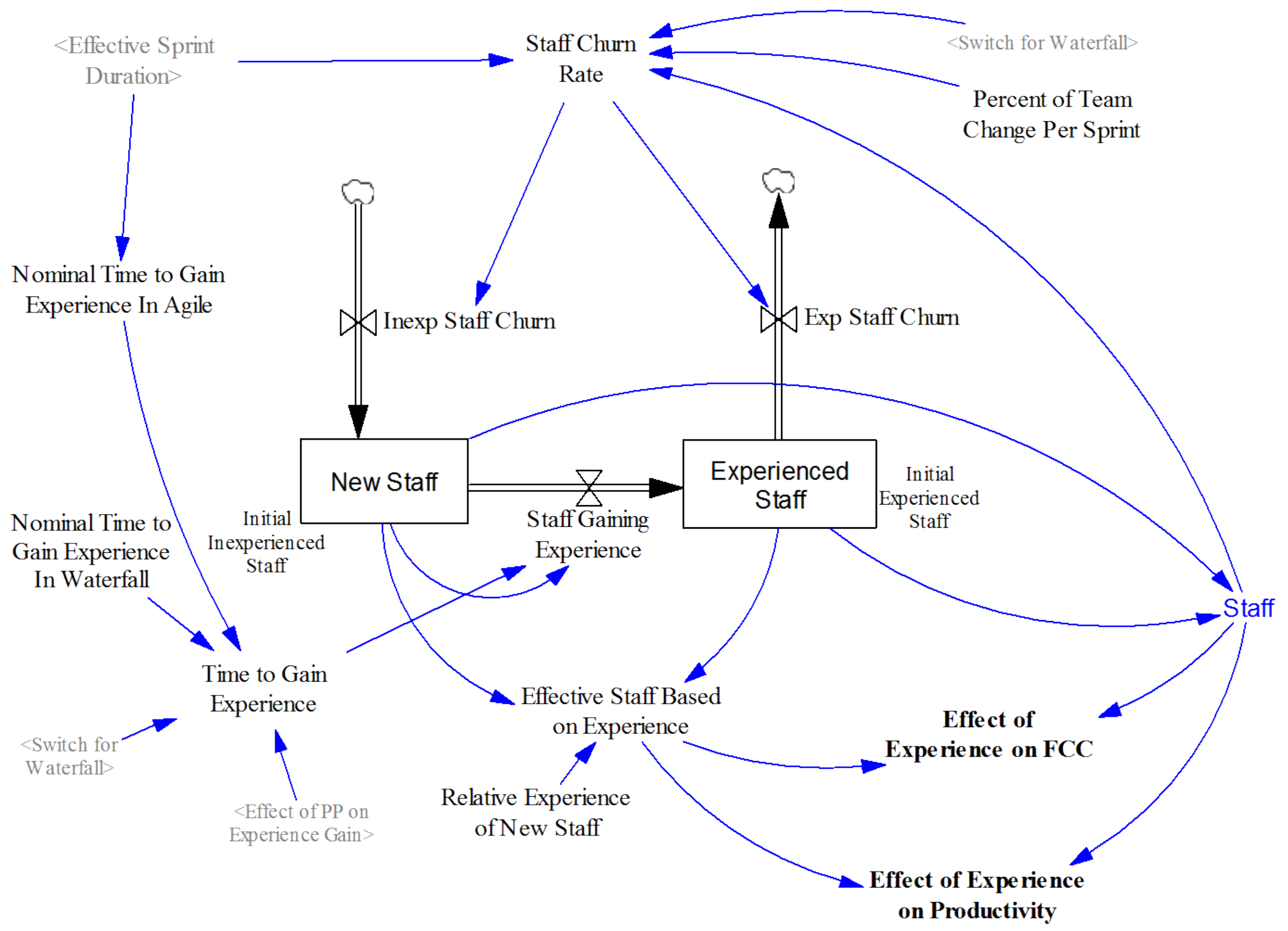
8.5 View 5: Software Integration and Test Cycle


Enable Integration and Test Activities
<Product Backlog>
<Release Backlog>
<Sprint Backlog>
<TIME STEP>
Project Finished
<InRelease>
Rework Discovery Rate from Int and Test
<Nominal Productivity>
Number of Integration and Test Engineers
<Undiscovered Rework in Release>
<Effect of CM Environment on Rework Discovery>
Integration and Test Productivity
Nominal Integration and Test Productivity

8.6 View 6: Continuous Integration

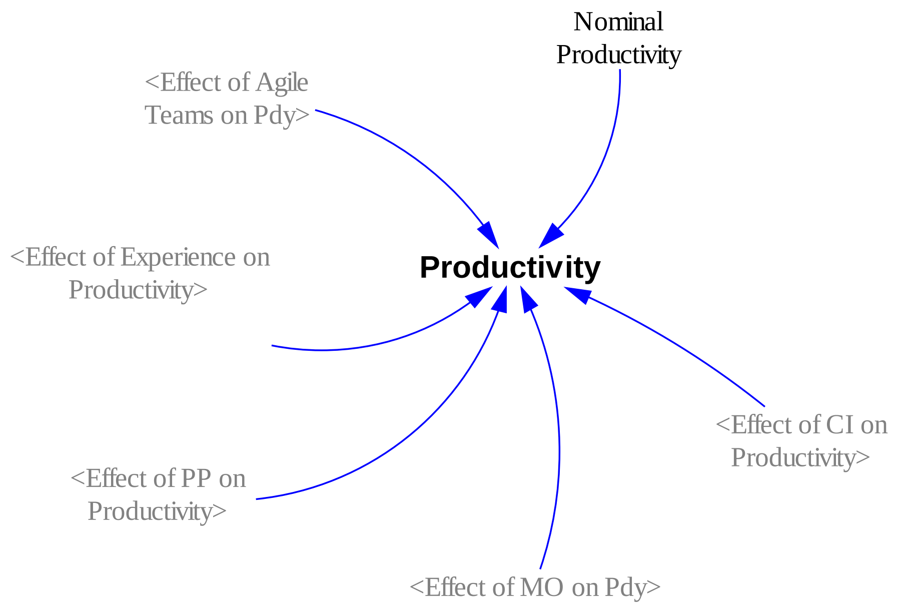

Allow Continuous Integration
CI Setup Work Generation Rate
<Current Release ID>
<InRelease>
<Current Sprint ID>
<TIME STEP>
Continuous Integration Setup Work
CI setup work rate
Time to set up CI
Number of Tasks to Setup CI Environment
Staff to set up CI
<Nominal Productivity>
CI Environment Available
Effect of CI on Productivity
Effect of CI on Rework Discovery Rate
Effect of Automated Testing on Rework Discovery
Effect of CM Environment on Pdy
Effect of CM Environment on Rework Discovery
Effect of Test Automation on Pdy
Level of Automated Testing Used
CM and Build Environement Automation Level

8.7 View 7: Staffing and Experience

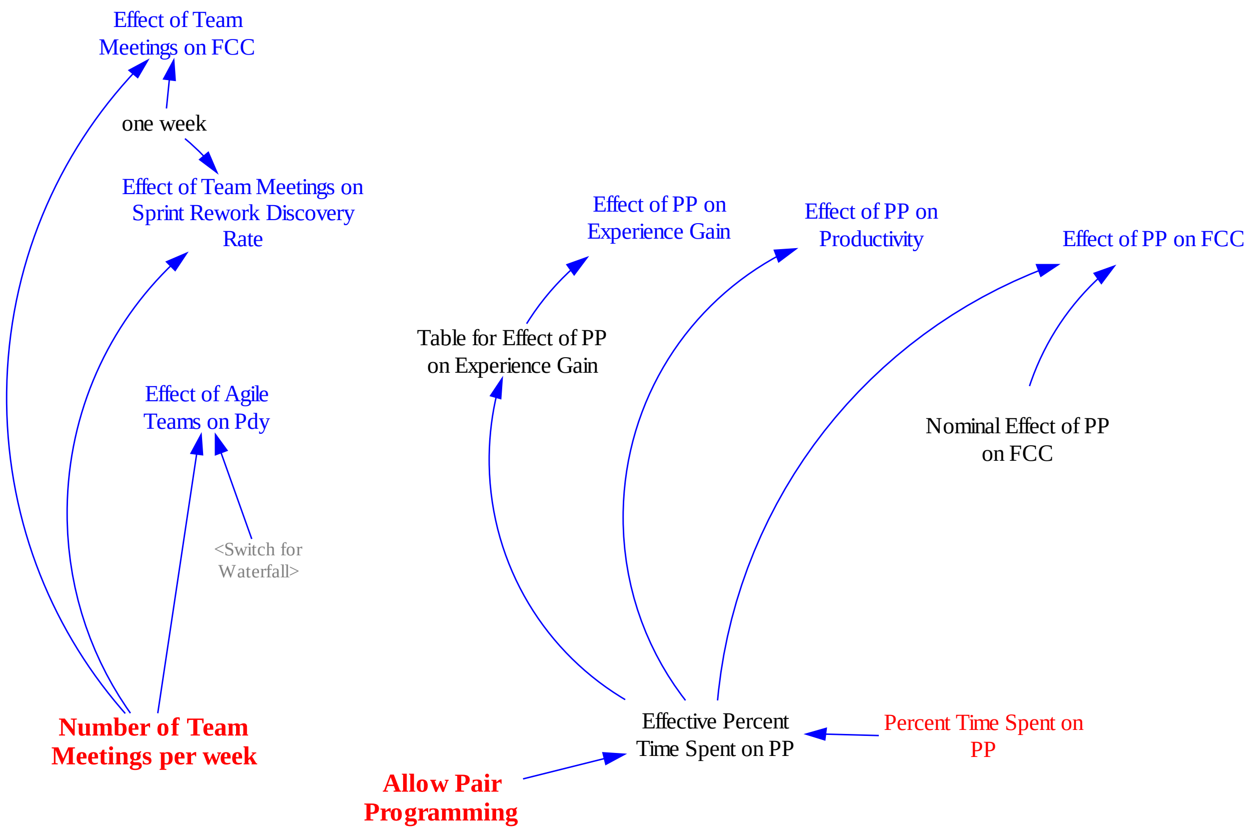


<Effective Sprint Duration>
Staff Churn Rate
<Switch for Waterfall>
Percent of Team Change Per Sprint
Nominal Time to Gain Experience In Agile
Inexp Staff Churn
Exp Staff Churn
New Staff
Experienced Staff
Staff Gaining Experience
Initial Inexperienced Staff
Initial Experienced Staff
Nominal Time to Gain Experience In Waterfall
Staff
Time to Gain Experience
Effective Staff Based on Experience
Effect of Experience on FCC
<Switch for Waterfall>
<Effect of PP on Experience Gain>
Relative Experience of New Staff
Effect of Experience on Productivity

8.8 View 8: Productivity

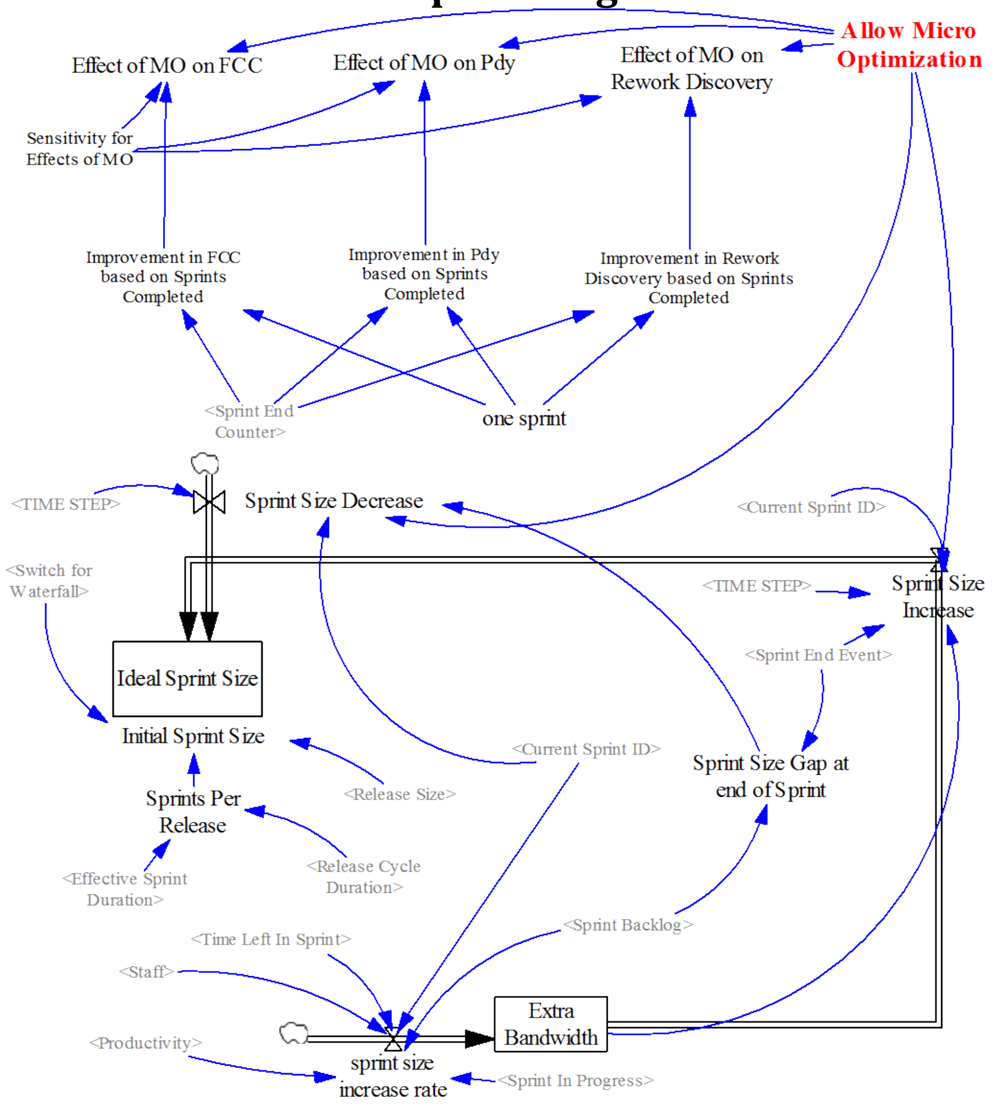

Nominal Productivity
<Effect of Agile Teams on Pdy>
<Effect of Experience on Productivity>
Productivity
<Effect of CI on Productivity>
<Effect of PP on Productivity>
<Effect of MO on Pdy>

8.9 View 9: Team Dynamics

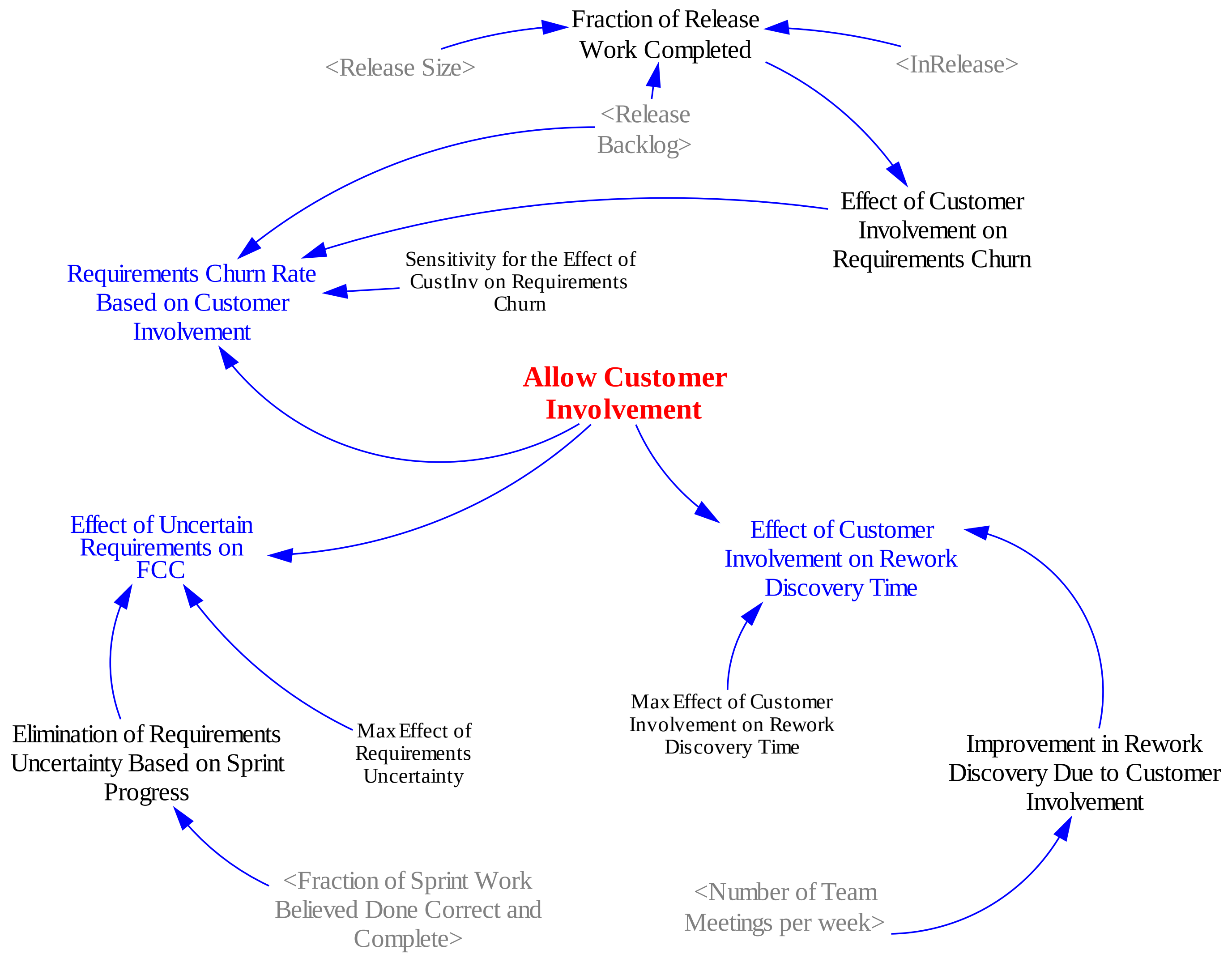


Effect of Team Meetings on FCC
one week
Effect of Team Meetings on Sprint Rework Discovery Rate
Effect of Agile Teams on Pdy
<Switch for Waterfall>
Number of Team Meetings per week
Effect of PP on Experience Gain
Effect of PP on Productivity
Effect of PP on FCC
Table for Effect of PP on Experience Gain
Nominal Effect of PP on FCC
Effective Percent Time Spent on PP
Percent Time Spent on PP
Allow Pair Programming

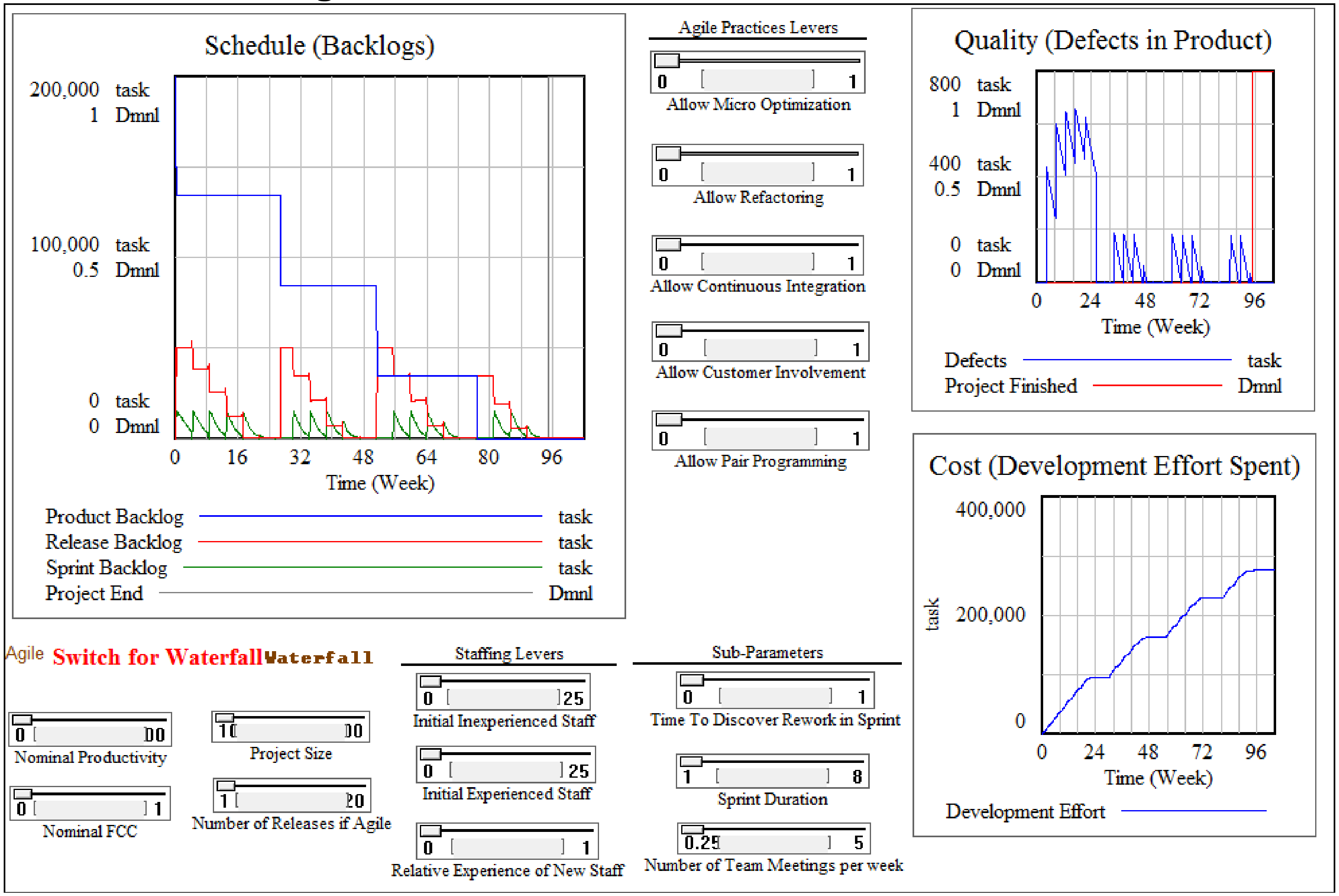
8.10 View 10: Micro-Optimizing



8.11 View 11: Customer Involvement


Fraction of Release Work Completed
<Release Size>
<InRelease>
<Release Backlog>
Effect of Customer Involvement on Requirements Churn
Requirements Churn Rate Based on Customer Involvement
Sensitivity for the Effect of CustInv on Requirements Churn
Allow Customer Involvement
Effect of Uncertain Requirements on FCC
Effect of Customer Involvement on Rework Discovery Time
Elimination of Requirements Uncertainty Based on Sprint Progress
Max Effect of Requirements Uncertainty
Max Effect of Customer Involvement on Rework Discovery Time
Improvement in Rework Discovery Due to Customer Involvement
<Fraction of Sprint Work Believed Done Correct and Complete>
<Number of Team Meetings per week>

8.12 View 12: Management Dashboard



Agile Practices Levers

Allow Micro Optimization 0 1

Allow Refactoring 0 1

Allow Continuous Integration 0 1

Allow Customer Involvement 0 1

Allow Pair Programming 0 1

Agile **Switch for Waterfall** Waterfall

Nominal Productivity 0 [illegible]

Nominal FCC 0 1

Project Size 10 [illegible]

Number of Releases if Agile 1 20

Staffing Levers

Initial Inexperienced Staff 0 25

Initial Experienced Staff 0 25

Relative Experience of New Staff 0 1

Sub-Parameters

Time To Discover Rework in Sprint 0 1

Sprint Duration 1 8

Number of Team Meetings per week 0.25 5