BUILDING SOFTWARE AT SCALE: UNDERSTANDING PRODUCTIVITY AS A PRODUCT OF SOFTWARE ENGINEERING INTRINSIC FACTORS

Gauthier Ingende Wa Boway
gingende@students.kennesaw.edu

BUILDING SOFTWARE AT SCALE: UNDERSTANDING PRODUCTIVITY AS A PRODUCT OF SOFTWARE ENGINEERING INTRINSIC FACTORS

A Thesis Presented to
The Faculty of the Software Engineering Department

By

Gauthier Ingende Wa Boway
BSc Computer Engineering, Mercer University, May 2024

In Partial Fulfillment

of Requirements for the Degree

Master of Science in Software Engineering

Kennesaw State University

May 2024

BUILDING SOFTWARE AT SCALE: UNDERSTANDING PRODUCTIVITY AS A PRODUCT OF SOFTWARE ENGINEERING INTRINSIC FACTORS

Approved:

Hassan Pournaghshband

Dr. George Markowsky

Paola Spoletini

Gauthier Ingende Wa Boway

Gauthier Ingende Wa Boway
311 Boulder Run
Hiram GA 30141

Dr. **Reza Meimandi Parizi**
Kennesaw State University
Marietta Campus
Office J 353D

BUILDING SOFTWARE AT SCALE: UNDERSTANDING PRODUCTIVITY AS A PRODUCT OF SOFTWARE ENGINEERING INTRINSIC FACTORS

An Abstract of

Thesis Presented to

The Faculty of the Software Engineering Department

By

Gauthier Ingende Wa Boway
BSc Computer Engineering, Mercer University, May 2024

In Partial Fulfillment

of Requirements for the Degree

Master of Science in Software Engineering

Kennesaw State University

May 2024

During our education at KSU, we have learned about various factors that affect productivity such as schedule, budget, and risks, but those are often controlled outside of what we could learn as software engineering principles, patterns, or practices. On top of that, other off-work factors such as health conditions, emotional distress, or political climate, just to name a few, could drastically affect the productivity of a software engineering team. We see a demarcation between those factors that affect productivity in software engineering but are not inherent to the discipline itself, which we call resistance factors, and the factors that are inherent to the discipline and drive productivity, which we call intrinsic factors. Intrinsic factors are driven by the way we learn software engineering in schools and the way we practice it in industry. During the master's coursework in software engineering, we identified three intrinsic inputs that play a solemn role in how we build and deliver software at scale: **people**, **processes**, and **tools**.

This thesis first provides an enriched understanding of each of the 3 factors listed above. It is an essential step to establish contextual reasoning about those factors before diving into deeper discussions concerning how those factors can drive software engineering productivity. A ***software engineering environment*** is the assembly of a team of people producing software following a defined set of processes and leveraging a specified set of tools. In a particular ***software engineering environment*****,** each of those inputs will exist in some state, and those states will interact with each other to realize a ***nominal productivity*****;** that is the raw potential of software to be effectively produced at scale in that environment accepting all other factors are equal elsewhere. To borrow electrical engineering language, it is the ***open-circuit voltage*** of software engineering team. That is the productivity potential that would have been attained if there were no resistance and zero risks, which we call here the ***nominal productivity.***

A ***nominal productivity*** framework exhibiting the interactions of those intrinsic inputs' states will be introduced to help analyze and understand their effect on productivity. We will then conduct some theoretical experiments with this framework and review the theorical findings with industry and academia experts to evaluate the fidelity of the framework.
We warn here that the goal of this thesis is not to validate the framework itself, and we recommend readers not to assess the quality of the paper by the validity of the framework introduced, but the scientific approach in answering the research questions and any advances that may fall from it.

The framework introduced, if nothing else, would serve as tool to software engineering managers to evaluate their current ***software engineering environment*** and prioritize the factors that need the most investment to build an effective and high performing team.

BUILDING SOFTWARE AT SCALE: UNDERSTANDING PRODUCTIVITY AS A PRODUCT OF SOFTWARE ENGINEERING INTRISIC FACTORS

A Thesis Presented to

The Faculty of the Software Engineering Department

By

Gauthier Ingende Wa Boway
BSc Computer Engineering, Mercer University, May 2024

In Partial Fulfillment

of Requirements for the Degree

Master of Science in Software Engineering

Advisor(s): Dr Hassan Pournaghshband

Kennesaw State University

May 2024

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

It is difficult to overstate the importance of productivity in software engineering in a world where software has become an undeclared necessity of everyday life. In 2011, Marc Andreessen, co-founder of Netscape and investor in Silicon Valley venture capital firm, warned that "*software is eating the world* (Andreessen, 2011)." He explains that this software invasion was enabled by the computer revolution 6 decades earlier followed by the invention of microprocessors in the 70s which preceded the internet boom in the 90s. Several years before cloud computing and smartphones with high-speed internet, all the technology required to shift businesses into a software centric model were already at play. Andreessen called out that "s*oftware is also eating much of the value chain of industries that are widely viewed as primarily existing in the physical world* (Andreessen, 2011)". Amazon, where virtually anything can be bought online and delivered to one's home was the main example to cite at the time; today we have Uber, Expedia, Webex, Dash, Waze, or Salesforce just to name a few software driven businesses. Even before COVID-19 struck, we were already partially living in a virtual world, the pandemic just made it official. One thing the pandemic did bring is the need for creating increased software to power that digital universe.

With that growing demand for software comes an amplified demand for software developers. Unfortunately, the U.S labor market is not supplying enough individuals capable of filling software development roles. In the third quarter of 2019, close to a million IT jobs positions were left unfilled (Loten, 2019). In a viewpoint published in the

July 2021, the ACM also adds that "*the pandemic is also taking its toll on the upstream supply of IT labor, and software developers in particular* (Moritz, 2021)". The journal explains that the number of international students, who make up a good proportion (35,200 out of about 136,000 in 2019) of awarded Computer and Information Science (CIS) degrees reduced by 43% during the pandemic. This can only accentuate the existing shortage in software developers. Many companies today rely on outsourcing some of their software development activities due to the low supply of this expensive skill. Moreover, the demand for software development jobs is estimated to grow by 22% from 2020 to 2030 (Bureau of Labor Statistics, U.S. Department of Labor, 2021).

Overcoming the 21st century world challenges, including pandemics, natural disasters, wars, and globalization require us to produce more software driven solutions. With the limited number of software developers on task, we cannot afford to lose more productivity due to poor management practices. Moreover, the making of software systems requires good software engineering management to be effective. There is a lot being done right now to drive up productivity in the software engineering industry. Most of the efforts focus on traditional management practices such as talent development, competitive compensation, retention programs, and work-life balance. This paper takes a stab at an organic approach by understanding the intrinsic factors of software engineering and how their interactions affect productivity.

RESEARCH QUESTIONS AND METHODOLOGY

Software engineering managers are always on a quest to make their team more productive. If we could read their mind, we certainly find the following question bold and underlined in font-46 title 1 heading:

"*How can I get them to hit those unrealistic deadlines upper management deemed highly critical to a key performance indicator thus existential for my next bonus?*"

Most of the time, the right answer is not making them work harder. Adding more people to the work does not solve the problem either (Brooks, 1982). Furthermore, to achieve a certain level of productivity, one needs to measure it, and that proves to be an even more complex task. How to compare productivity from one team to another when ground realities are not the same. What metrics to consider and what tools or techniques are appropriate for measuring those metrics? Those questions are not simple to answer. On top of that, Heisenberg's principle of observation applied to software engineering productivity measurement would mean that measuring productivity would affect the productivity itself. The classic example is those Agile boards that a respectable number of software shops use. Software engineers must spend time maintaining those boards so that productivity metrics like sprint velocity or burn down rate are captured.

Worse even, software engineering productivity measurement comes after the fact, and there is not always time to catch that competitor who has announced the release of their brew of the same product next week.

Considering the preceding, we deemed it important to explore the following questions for our research:

<u>Research Question 1:</u>

Among all the factors that can affect productivity in software engineering, which ones are inherent to software engineering common principles, practices and patterns?

<u>Research Question 2:</u>

How do practitioners see the interactions between people, processes, and tools affect productivity in software engineering?

<u>Methodology:</u>

In this research exercise, we will focus on the intrinsic inputs of software engineering productivity we learned from the coursework in the Master in Software Engineering program at KSU.

From that analysis, we will derive a productivity visualization model that would allow us to estimate ***the nominal productivity*** of the ***software engineering environment***. The model will allow us to run some theoretical experiments in which we will vary the state of each input and see its impact on nominal productivity.

After we conduct the theoretical experiments, we plan to present the findings to experienced practitioners to gather some feedback on whether the scenario described in each experiment is like what they have encountered in their journey and on how effective the framework is in explaining the observed change in productivity.

As stated before, we warn that the goal of this thesis is not to validate the model introduced but to explore the possibility of a sensible visualization model that overlaps the concepts we have learned during our coursework with their impact on productivity. The research will take form in a questionnaire that we have designed, and it will be executed by a recognized third-party research firm. Due to the nature of our research, subjects' recruitment has proven to be a challenge, and we were only able to secure 30 participants. For deeper insights into the research, we would need more financial resources than available to us at the time of this writing.

CONTEXTUAL DISCUSSION ON INTRISIC PRODUCTIVITY FACTORS

1.PEOPLE

§ If we ever had a talk with any genuine engineering manager, they would tell us that human factor is the most impactful factor during any engineering project in any engineering discipline. In a quest to better understand the 3 intrinsic inputs of software engineering productivity, we will start with **people.**

There are numerous psychological, anthropological, and sociological discussions about **people**. We do not plan to add to that set; here, we want to discuss **people** in context of what happens in software engineering trenches. Nevertheless, to balance the conversation, we may refer to some generally accepted psychology theories. In software engineering context, there are three attributes of **people** we found significant: **technical skillset**, **personality,** and **organization**.

Skillset

While skillsets are required for any type of labor, it's important to note the multitude, volatility, and spread of skills that are involved in the building of software systems. From conversations with practicing software engineers, we created the following word cloud that shows the spread and frequency of usage of technical skills they use depending on the task at hand:

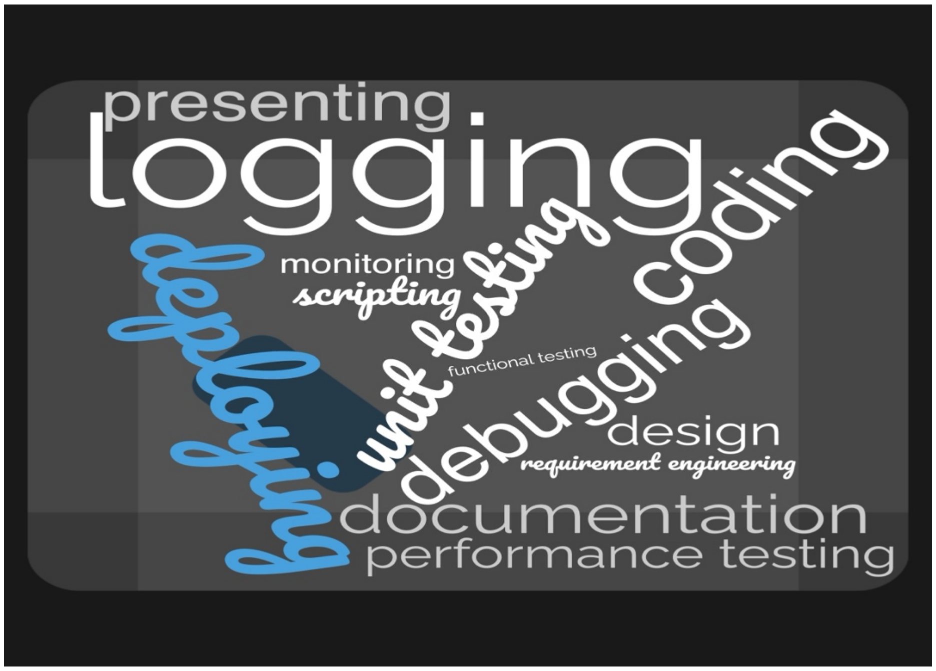


Figure 1: Software Engineering Skills Multitude and Volatility

As we can see, people in software engineering go back and forth between a litany of skills to efficiently perform their jobs. Moreover, with the advent of Agile practices and concepts like full-stack engineering, software engineers are expected to participate in all phases of the Software Development Life Cycle. While in most other engineering discipline specialists are wanted and acclaimed, they are less and less desired in software engineering in favor of people who are willing and able to expand their skillset. This poses as a challenge for people who want to learn one thing and apply it during a long span of time to become experts. Also, while large scale software projects are executed by a group of people, it does not mean that everyone participating equally contributes in terms of time and effort in such a way we can switch people between tasks and obtain the same output.

Personality

We hire people mainly for their skills; but when people come with their skills, they also bring their personality along. To borrow again from electrical engineering, we will think about the signal and noise concept. When an electrical device generates a signal that contains useful information, it also generates some undesired signal (hear noise here) which is inherent to its internal properties. While building electrical circuits, the goal is to maximize signal strength and minimize noise strength. In software engineering the same effect is also desired between people's skills and their personality, but the task here is much more difficult. We have seen managers try to exchange people with similar skillsets across teams in motion and expect to maintain the same productivity. The outcome of the experiment was shockingly disappointing.

Apart from technical skills, there is another set of intangible abilities called "soft skills" that people bring to the team. These are often a good indicator of a person's personality and play a significant role in how people approach work and react to the challenges of the work environment.

In her work discussing the conflict between personality and skillset required to succeed in the modern, Bonnie Urciuoli retakes this passage from Menochelli's work:

<< *A soft skill refers to the cluster of personality traits, social graces, facility with language, personal habits, friendliness, and optimism that mark each of us to varying degrees* (Urciuoli, 2008)>>

Urciuoli adds that soft skills represent an imaginary line between self and work.
From that angle, personality can be understood as the result of the collision between self-interest and job-interest, and it drives how one interacts with the task at hand. One question that arises here is, when confronted with a judgement-based decision point, do we rely more on our inner valuing systems, or do we leverage more of the surrounding factors at play?
This thinking aligns well with Carl Jung's extrovert vs introvert personality theory. He defines personality as continuum between introversion and extroversion. The first is a trait found in people who are often motivated by self-internal factors while the latter is found is people who are more stimulated by external environmental factors. In 2013, Susan Cain introduced ambiversion which serves as a midpoint between the two extremes (Cain, 2013). The diagram below, extracted from a summary of the book found online, does an excellent at depicting this idea of an introvert-extrovert personality continuum (Academic Accelerator, Mar).

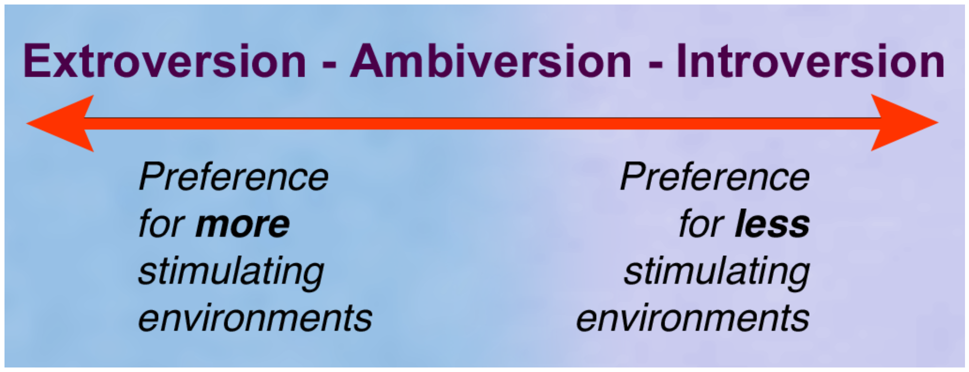

Extroversion - Ambiversion - Introversion
Preference for more stimulating environments
Preference for less stimulating environments

Figure 2: Extroversion Continuum

On top of the Jung's theory explained above, American personality psychologist Golberg R Lewis created a superset of personality traits named the "*Big Five*" (Goldberg, 1990). He also lists introversion as a personality measure but introduces four additional dimensions of personality: openness, consciousness, agreeableness, and neuroticism. While these new dimensions add more details and color to people's personalities, we will limit our upcoming analysis to extroversion, as this dimension has far greater interaction with the ***software engineering environment***.

Organization

We have hired a group of skilled people, and with their skills they also brought on a personality which we understand better now even though we may not like it. The next challenge is how do we make all these people work together.

First, we said earlier that large scale software is built by a group of people. We will take that back. We meant large scale software is built by a team of people.

So, what is the difference between a group and a team as it relates to people?

A team is a circle of organized people who work together to fulfill a common mission which also resonates with everyone's personal goals. Before the team is created, the mission must exist, and anyone who joins the team must participate in achieving the team mission. While it is true that most folks will say that the primary reason why they show up every day at work is to get paid, some software engineers perform beyond expectations because they believe in and enjoy what they do. Salary is not a motivation; it is a compensation. Engagement creates motivation, and we create engagement by defining a mission all team members want to be part of. In 2013, 70% of American workers were reported not working to fullest potential because they are not sufficiently engaged causing an estimated $550 billion every year to American company (Gallup, 2013). The Global analytics firm Gallup defines employee engagement as “the involvement and enthusiasm of the employees in both their work and workplaces (Gallup, 2013).” In lay man terms, the question here is “are the people in your team proud of what they are doing? If not, there will be some productivity losses.

We need to define an engaging mission for the team, and once we do that, the next thing is to establish some type of order. While cost varies with the addition of people into a project, progress does not, unless all tasks are partitioned in such a way that there is no communication required between people working on the project (Brooks, 1982). Let us be honest, this never happens in large scale software engineering.

A team structure is created by the definitions of roles, responsibilities, and relationships. Fred Brooks addresses this well in chapter 3 of his classic. He suggests we create “*surgical teams*” where every person has a rigorously defined set of attributions and performs those better than anyone. As the French saying puts it, let every man (or woman, in the modern industry) do his job, and the cattle will be safe.

At this point, I can hear the Agile purists scream to the top of their voices that this should not be so. They suggest we create "*balanced teams*" where everybody is equally capable of taking on the task at the top of the backlog. We also agree with the idea, and we even tried it in our many years down in software production trenches, but the beauty of this theory was betrayed by the realities of the field. It is very hard to find a team where everybody can perform the same task at the same level of efficiency. Fred's "surgical team idea" rests upon research done by Spackman, Erikson, and Grant that concluded that there is an order of magnitude observed in productivity between high performers and low performers for given set of tasks; since this whole thesis is about productivity, this finding is of high significance (Sackman, Erikson, & Grant, 1968). The idea is that even though we want versatility so that we can reduce dependency and achieve sustainability, we must consider the ground reality of disparate individual capability and make them work for our advantage. If we sacrifice tactical planning and assignment and decide to pull from the top of the backlog like Agile priests preach us, we will gain sustainability, but we will lose speed.

In a world where software engineers are in high demand and speed the market is fatal for business, many are the times when speed is favored over sustainably because of tight business constraints. The choice here cannot be dogmatic; we must be pragmatic and play it by ear. The type of team structure suited for a team depends on the nature of the work being done (Mantei, 1981). In her work, Marylin Mantei compares the effect of utilizing Weinberg's "***Chief Programmer Team"*** approach versus Baker's "**Egoless *Team"*** approach over programming tasks of varying difficulty, size, duration, modularity, reliability, deadline penalty, and sociability. The key result is that even though both approaches had varying success here and there, neither of them proved to be better than the other overall. The appropriate type of team structure depends on the characteristics of the task being worked on.

Another challenge that arises when we put a team of people to work is to establish the appropriate workplace relationships. Some workplaces interactions are task-driven while some others are relationship-driven. The diagram below maps the appropriate relationships to develop to the dominant need type of the people on the team is shown below (Dent & Brent, 2015) .

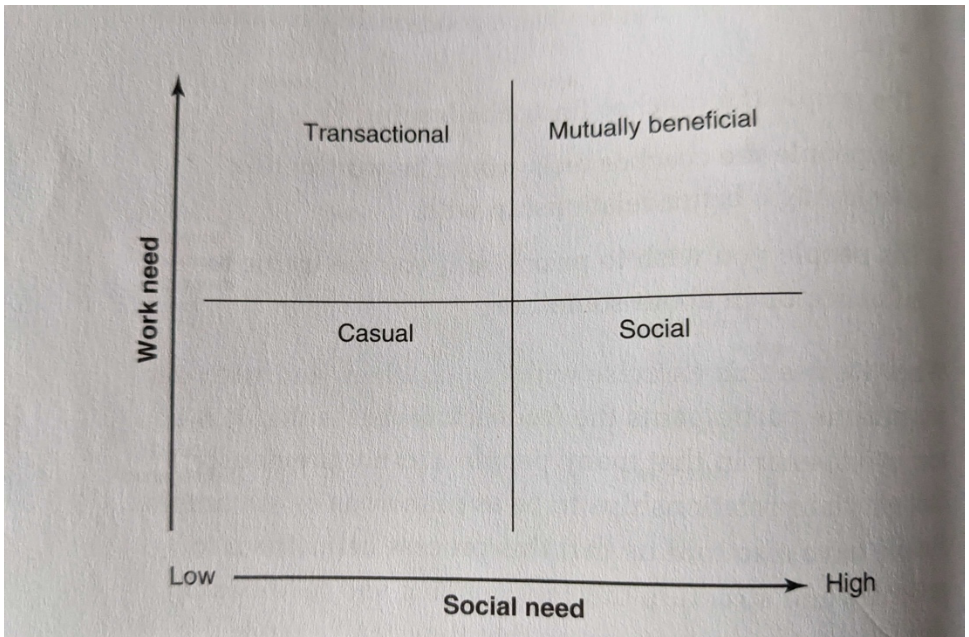


FIGURE 16.12 Work relationship analysis model

Figure 3: Work Relationship Analysis Model

A **transactional** relationship is a relationship in which people work together but choose not to socialize. **Casual** relationships are those where there is little professional interaction and little socialization. On the other hand, **mutually beneficial** relationships have very high levels of professional interaction and socialization. The last quadrant is for **social** work relationships where there is little professional interaction but a high level of socialization.

In the length and breadth of this paper, we will not be able to exhaust all the aspects of workplace relationships. The point here is to raise awareness of the existence of those nuances so that software engineering managers do not fall into the trap of making homogenous relationships assumptions. Awareness of those nuances will help software engineering managers find the optimal balance to guarantee smooth collaboration.

2.PROCESSES

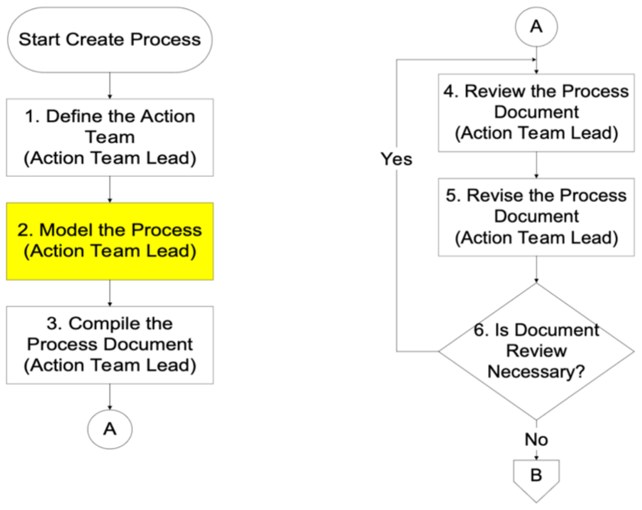
§ Everyone recognizes the need for competent, motivated, and diversity-aware people for software engineering teams, yet those very same people will perform poorly if provided unclear and insufficient directions.

Processes help us define what are the expected inputs and outputs of any task, and what are the sequence of steps that need to be followed to transform those inputs into outputs. Processes also guarantee that the same input into a task will always produce the same output given that all pre-conditions are met. This is key in achieving consistency with a team with many people with disparate personalities and competency levels coupled with a diversity of culture; the process is really the common denominator of a team, and it helps everyone move forward in the same direction.

Let us face it, companies have priorities and initiatives; people have agendas, which are mostly made up of bullet pointed selfish goals they wish they achieve before they are forty, or before they retire for those who did not achieve them by forty. There is always inconsistency in the way a large enough number of people accomplish the same task. One way we solve this is to establish some rigorous processes that everyone must follow while achieving anything meaningful to the company. We have mentioned processes a few times in this section, but strictly speaking, what can be considered as a process in software engineering?

A process is what people do using procedures, methods, and equipment to transform inputs into output that is of value to customers. It is a framework to accomplish tasks with the high chances of attaining the desired output while respecting the predefined constraints. Speaking of frameworks, software engineers understand very well that they are scaffolding into building software in a safe and repeatable manner. A process should be targeted to solving the problem of facilitating a task or set of tasks. This assumes a thorough understanding of the task by the person who is creating the process around it. Unfortunately, after working in the industry for close to nine years now, we know this is not always the case. Numerous are the times where the software engineer manager or the lead engineer will ask somebody to do something they do not even understand themselves, yet they expect the person to have it "done well." What is the definition of "done" for the task, and what is the definition of "well." Failure for the task requester to precisely answer those questions can yield some heated arguments and confusion when deliverables are submitted. A process acts like a contract between the task requester and the task performer. It removes the ambiguity on how to perform the task and creates a precisely defined rendezvous point for the outcome of the task.

Nevertheless, while processes shed light on the path to success, that light itself must be bright enough to show enough of the path. That is, the process itself must be comprehensible and usable. The flow chart below proposes a step-by-step guide on how to create effective processes (Pournaghshband, 2021).


Start Create Process
1. Define the Action Team (Action Team Lead)
2. Model the Process (Action Team Lead)
3. Compile the Process Document (Action Team Lead)
A
A
4. Review the Process Document (Action Team Lead)
5. Revise the Process Document (Action Team Lead)
6. Is Document Review Necessary?
Yes
No
B

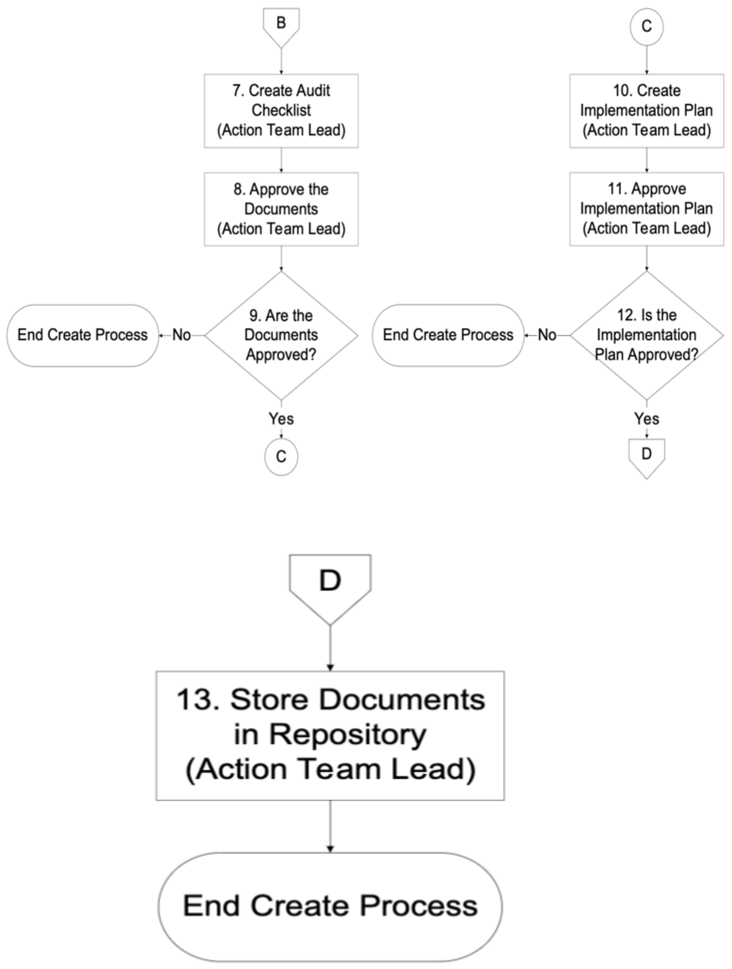
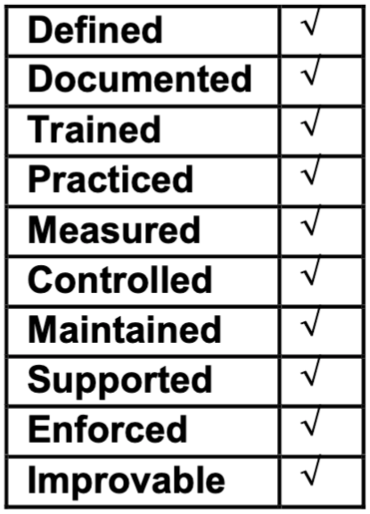

B
7. Create Audit Checklist (Action Team Lead)
8. Approve the Documents (Action Team Lead)
9. Are the Documents Approved?
No
End Create Process
Yes
C
C
10. Create Implementation Plan (Action Team Lead)
11. Approve Implementation Plan (Action Team Lead)
12. Is the Implementation Plan Approved?
No
End Create Process
Yes
D
D
13. Store Documents in Repository (Action Team Lead)
End Create Process

Figure 4: Effective Process Creation Guide

§ This guide above aims at providing a chain of business benefit that goes from improved schedule and budget predictability to improved employee morale which induces increased productivity and quality which brings increased customer satisfaction, which eventually causes increased return on investment.

On the other hand, processes must be measured for effectiveness and evolve with time to cope with ever-changing business needs. The table below illustrates the progressive state of a process maturity (Pournaghshband, 2021).



Defined	√
Documented	√
Trained	√
Practiced	√
Measured	√
Controlled	√
Maintained	√
Supported	√
Enforced	√
Improvable	√

Figure 5: Process Maturity Scale

From the table above, we can observe three main phases in the ***process maturity scale*** of a process. The definition phase regroups the first two states and is the vital phase of process maturity evolution. As we mentioned earlier the process must first exist, and it only exists if it is in writing, or in today's digital words in some form of digital documents. Failure to have formally documented processes will create an immature organization with no reliable approach to repeating challenges coupled with inconsistent results and inadequate value growth.

The implementation phase of the *process maturity scale* combines the "*trained*," "*practiced*," and "*measured*" states and represents a major milestone in the process evolution. The process audience needs to be aware of the process and practice it. It is important to note that this will probably be the most time-consuming phase for any large enough organization. This is also where we can make measurements for the effectiveness of the process. Measuring the process is critical as it helps not only to evaluate the value of the process itself, but also the value change in the product being output by the process. Processes are made to improve return on investment. It takes a long time and a lot of resources to change people behavior or force them to operate following a new process, so each process must go through all the maturity state to be proven to yield the expected improvement or it would surely yield unsatisfactory results by decreasing both employee morale and the return on investment for process stakeholders. Practice makes perfect, but practice takes time, a lot of time. Impatient process stakeholders will never reap the benefits of the process if they fail to invest the adequate amount of time and resources required to train the process audience to the point where they can autonomously execute the process.

The last phase which encapsulates the *"controlled," "maintained," "supported,"* and *"enforced"* states of the **process maturity scale** is the enforcement phase. At this point, we have "mastered" the implementation of the process, and we have proven that it yields productivity benefits, but we need to make that all are following marching orders. If a big organization with millions of dollars at stake, there is no tolerance for assumption; everything needs to be verified. Enforcement is a two-way street. It helps the leader ensure his vision is followed, but it also helps the subordinates ensure that they get the proper reward/recognition when they follow the vision.

The last state in the ***process maturity scale*** is *"improvable."* In practice, this state is really hidden behind all the other states in the scale. The maturity scale could be represented as a circle that we keep rotating around with each full turn being one level of improvement. This is a continuous activity that should be carried out on predetermined cadence by all process stakeholders.

3.TOOLS

So far in this discussion, we have hired a set of competent people that we formed into a team, and we have created processes to guide them in the tasks we assigned to them; now we need to equip the team will all the tools they need to efficiently do the job. If software engineering were warfare, tools are the weapon we would fight with. Since modern wars cannot be won with swords and spears, modern software calls for several advanced tools and technologies to produce, package and deliver software. The diagram below shows various tool types and which phase of the software development lifecycle they support.

Tool Function	Examples	SLCD Phase Supported
Integrated Development Environment	eclipse WS	CONSTRUCTION TESTING
Version Control		CONSTRUCTION TESTING
Programming Languages	GO HTML JS CSS	CONSTRUCTION TESTING MAINTENANCE
Code Analysis	sonarqube	TESTING
CI/CD	Concourse Jenkins TC TeamCity JFrog ARTIFACTORY	CONSTRUCTION
Monitoring:	splunk> dynatrace ZIPKIN	MAINTENANCE
Communication	O	ALL PHASES
Task Management	Pivotal Tracker Jira	ALL PHASES

Figure 6: Tool/Technology and corresponding Software Development Lifecycle phase.

In context of software engineering, a tool is a computer program that helps automate a task executed by a machine. Twenty-first century large-scale software development, which requires the execution of an almost infinite number of tasks, has been enabled by several advances in automation we have seen accelerate in the last two decades in the industry. The race for automation has fueled a proliferation of tools which occasioned the release of immature and sometimes counterproductive tools on the market. We have witnessed a tool that caused a critical task to fail and the company to lose money. Blue/green deployment was being implemented with a tool that was supposed to route incoming request traffic to a secondary availability zone while we were upgrading the software in the primary one. The tool failed to do so correctly, and this was not the first time. Incoming requests continue to come into the availability zone being maintained and caused some sales transactions to fail. The managerial response was to have a dedicated team monitor incoming requests traffic shaping during all deployments to make sure that the tool performed correctly. Creating a burdening process around an unreliable tool is paying twice for an item that was never delivered. Just like the people who are hired; a tool must meet certain criteria before the team adopts it. We recommend software teams to conduct a thorough analysis based on three criteria: mechanical advantage, accuracy, and consistency. A tool eligible for adoption must yield all three benefits.

The mechanical advantage is obtained through speed and fatigue. The tool must perform the task much faster than people and must do so continuously without rest. One fitting example that comes to mind is version control tools. Yes, we can keep track of changes made to a set of files and decide what state of the files I want to keep, but how many files can keep track of in a day, and what if those are modified multiples time by more than 20 plus people in that same day. Obviously, a tool will do a much better job than us here.

The second advantage a tool must offer is accuracy. People can do measurements and computations, but the probability of a human mistake in an overly complex measurement or computation is much higher than that of a machine. Think about calculating the unit test coverage of a piece of code that requires calculating the cyclomatic complexity of a method that has 14 “if statements” in while loop that with 5 flags condition. If we are building critical systems, we will better use a tool do that.

The last advantage a tool must provide is consistency. We said it before in this discussion; for the same task, different people will produce comparable results in separate ways.

A tool helps ensure that the task is executed systematically following a deterministic procedure, and since machines are far less prone to indiscipline than humans, we can expect consistent results.

Nevertheless, we must warn here that tools do not replace people, they amplify them. Machines are good at executing tasks fast and without rest, yet machines do not have any creative capability, and this is the single most reason people will always be used for building software. Their creativity is what makes them indispensable, but their behavioral and physical limitations create the need for tools. Tools compensate for human limitations during the execution of tasks, but as we discussed before, the appropriate tool must be used to reap the expected benefits.

INTRODUCING THE NOMINAL PRODUCTIVITY FRAMEWORK

1.PRESENTATION OF THE FRAMEWORK

Let’s provide a simple working definition of productivity: it is the ratio of output over input. It should be that simple, shouldn't it? Earlier in this thesis, we have taken some time to discuss the intrinsic inputs of software engineering productivity, but what are the intrinsic outputs? Is it the number of lines of code produced, is it the number of JIRA stories completed, it is the number of production deployments, is it the number of App store downloads or usage statistics in case of internal apps, or is it the amount of revenue generated by the application? Even though we have studied at the graduate level and practiced software engineering for a living during the past nine years, we still find it difficult to answer the question. Moreover, from reviews with others in academia and in industry, we found that software engineering productivity is hard to measure quantitively and objectively. This is illustrated by this passage from *The Pragmatic Engineer* blog:

<<*Try to measure any one dimension, and you'll fail. Measuring lines of code is meaningless and leads to busywork. Number of tickets closed? People will get creative opening tickets, or optimize for the easy-to-fix ones. Number of commits per day? It will lead to small and frequent commits, but not more. Any single metric you give to people, it can - and will - be gamed* (Orosz, 2022)>>.

Moreover, we need to distinguish activity from productivity. Most metrics we utilize today in the software industry are mere indicators of work being done, but is that work creating value for the business? While we can continue the enticing debate on which productivity metrics are representative and accurate, the American Analytics and Advisory firm Gallup finds that the biggest question is: "***What are the conditions that promotes productivity?*** (Maese & Robison, 2021)" **.**
This interrogation has inspired us to introduce the **nominal productivity framework**. The nominal productivity of a team is the potential of a team to produce software due to its internal setup and excluding the effects of constraints factors such as budget, dependencies, and schedule. Nonetheless, we imposed ourselves some constraints in the way the **framework** will behave to reflect how productivity behaves in real software development trenches.

From years of observation in the industry, we have noticed that productivity is typically not a continuous function, it is very discrete. The level of productivity between high-performing teams and the average team is sometimes an order of magnitude greater, but the difference in productivity between an average team and a poor team is abysmal. There is an imaginary threshold of team health that must be attained before unlocking any significant productivity. Below that threshold, teams are refractory to any productivity boost. Even adding more people does not help (Brooks, 1982). The foundational team setup issues must be resolved.

We showed earlier in this discussion that good team health rests upon defining a compelling mission, choosing the optimal structure, and developing the appropriate workplace culture. While building a team, the goal is not to maximize, but to balance. Productivity is not a monotonic function whereby we can drive output to the maximum by driving input to the maximum. Hiring a PhD skillset for a project that requires high-school-plus-some-bootcamp skillset will not maximize productivity, it will ruin it. Creating a very stringent process for people working in innovation-based roles will shy them off and make them unproductive.

To steal again from electrical engineering concepts, we can say productivity has a transient response and a steady state response. The transient is what we are interested in here. What is the right amount of boost we can give the team before we reach breakeven, and productivity does not follow the stimulus anymore.

The last key observation is that while we know that the first past to this exercise will not gain unanimous praise, our goal is to at least create the possibility of a sensible visualization model that overlaps the concepts we have learned so far with their impact on productivity. It is a framework one can use to explain and think about productivity based on how we understand and practice software engineering. Consider its layout below:

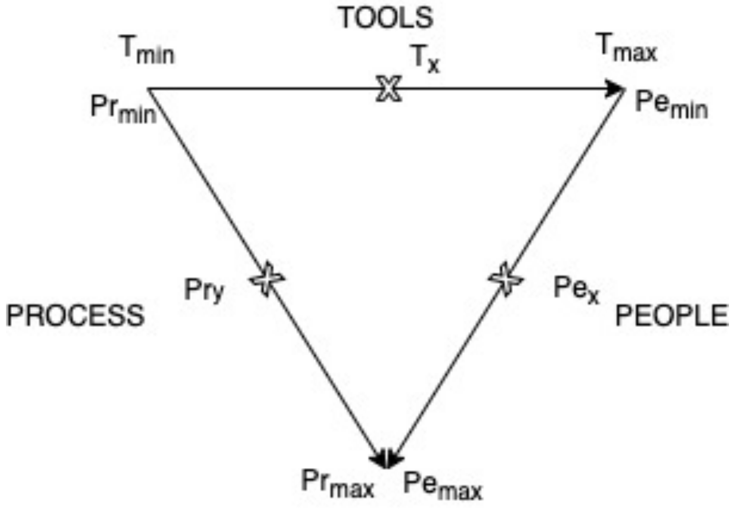


Figure 7: Nominal Productivity Framework

It is represented as a triangle in which each side is a directed and graded axis on which we plot the state of each of the three productivity inputs we are working with. The position of each point denotes the qualitative state of the input or the level of sophistication/maturity along its axis, and the surface area of the triangle cornered by these three plotted points denotes the nominal productivity achieved with that combination of input states. Let's plot an example to familiarize ourselves with the working of framework:

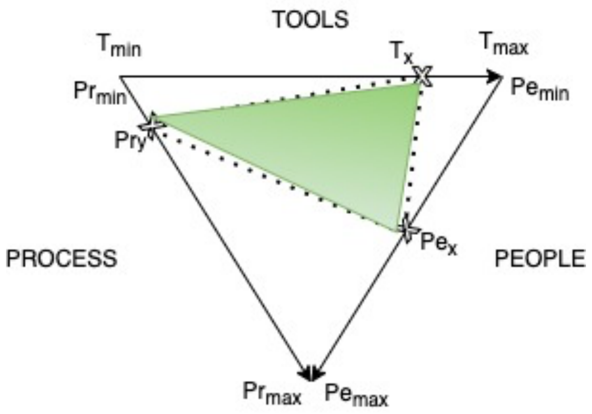


Figure 8 : Example Usage of Nominal Productivity Framework

In the example above, we plotted one point on each of the graphs. T_z is the level of sophistication of the tools we want to use in the team, which is high in this case. It might be a choice we made or an existing condition we found as we start putting together the development team. We hired a team of average people (Pe_x is about in the middle), and we decided to go with simple and rudimentary processes (Pry is close to Pr_{min}.). The area in green is the nominal productivity we get. The area of the big triangle is the maximum nominal productivity, which we may never get. The goal is to maximize the area in green by moving T_z, Pe_x, Pr_y but as we learned from the earlier discussion, there are implications in changing those input. This framework helps visualize and explain the impact of changing the states of each of the three intrinsic inputs of software engineering productivity.

As we move the plotted points in each axis of the triangle, the obtained nominal productivity must respect the following constraints:

1. Since productivity has discrete levels, they should a finite amount achievable level of productivity given the number of states for each factor. Each axis will have a finite number of graded states, which we will define later.
2. Since productivity has a transient response, the nominal productivity cannot keep increasing at we keep increase one input (qualitatively or quantitatively)
3. Since productivity does follow not monotonic function curve, lower position of certain input along their corresponding can produce bigger nominal productivity than the higher position, other inputs being constant.

2.QUALITATIVE GRADING OF THE FRAMEWORK INPUTS

At this point, we have established a working tool to visualize the change in nominal productivity and we move the states of its input. Observing the visualization model, we understand that reaching the high end on each input axis is not the goal here, as this will not yield the suggested nominal productivity. In fact, the first thing we learn from this framework is that the maximum nominal productivity is not obtained by combining Pe_{max} , Pr_{max} and T_{max} , that is maximizing investments in people, processes, and tools. Funny enough, it is equivalent to investing the minimum in each of those input. We can see that combination (Pe_{min} ,Pr_{min} and T_{min}) and combination (Pe_{max} ,Pr_{max} and T_{max}) both produce straight lines with surface area of 0.

The challenge here is to find the combinations of positions Pe_x, Pr_y and T_z, on each side of the triangle that will maximize the nominal productivity and confront the implications of those combinations with the explanations we provided earlier in this discussion. To effectively practice this exercise, we will need to grade each of the directed axis that make up the triangle. The number of positions we want to work with on each axis is subjective, as it does not change the working of the model; it just gives more possibility for experimentation with the environment conditions.

Let us start with “people” axis. We have discussed three key attributes of interest in software engineering: skillset, personality, and organization. In assessing people for hiring, we emphasize skillset, as this is the primary measure or whether the individual can fulfil the duties of the job. As we explained earlier from signal and noise analogy, personality and skillset are received in the same atomic transaction in we hire people. Organization is something we as software engineering managers establish by a mechanism of team which should have a mission, a structure, and a culture. For this to work effectively we dream of drafting a team of talented and homogenous people who are all modeling our culture (think the utopic no-drama team where everybody gets along super well). Unfortunately, we have not seen this happen in the industry; there is often a good jambalaya of various levels of personality which can conflict among one another and with the desired organization. We recall that our working definition of personality in context is the result of the collision between self-interest and job-interest. From these reasoning we can emerge the following possible categories for people:

1. The low skilled – strong personality vs organization fit category
2. The low skilled – good personality vs organization fit category
3. The low skilled – weak personality vs organization fit category
4. The medium skilled – strong personality vs organization fit category
5. The medium skilled – good personality vs organization fit category
6. The medium skilled – soft personality vs organization fit category
7. The high skilled – strong personality vs organization fit category p/o
8. The high skilled – good personality vs organization fit category
9. The high skilled – weak personality vs organization fit category

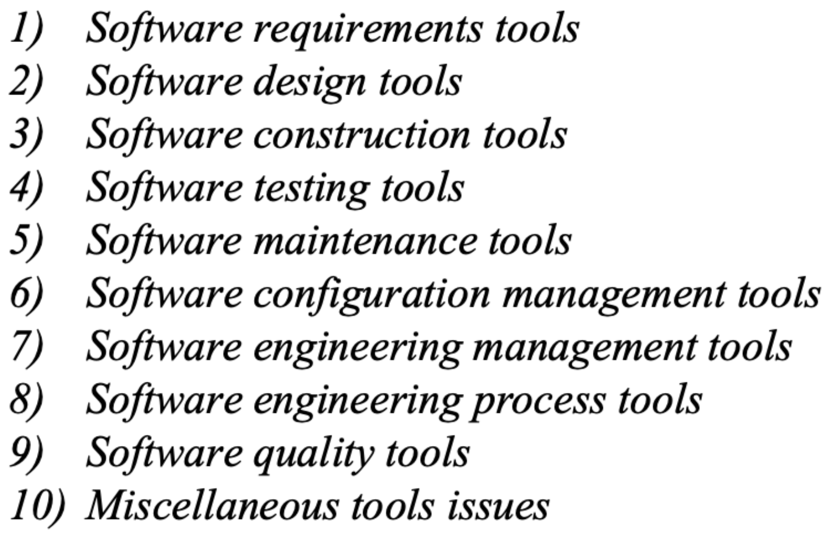
The first three categories are the ones we do not want to hire from. As we said earlier, people are hired primarily for their skills; if the skills are insufficient, the rest is irrelevant. We can then collapse category 1-3 to a single low-skilled category. We obtain these 7 simplified and effective categories of people.

1. The low skilled category
2. The medium skilled – strong p/o fit category
3. The medium skilled – good p/o fit category
4. The medium skilled – weak p/o fit category
5. The high skilled – strong p/o fit category
6. The high skilled – good p/o fit category
7. The high skilled – weak p/o fit category

The 3 medium skilled groups are where most people will be found with varying degrees of personality fit.

The highly skilled category is made up of extraordinarily talented people. Sometimes they are very easy-going with soft personalities which easily fit the desired organization or sometimes they are reluctant to change with hard personalities which conflict with the desired organization. The question now becomes whether we should put more weight on skillset or on personality vs organization fit. We will defer answering the question until we consult the expert later for our research.

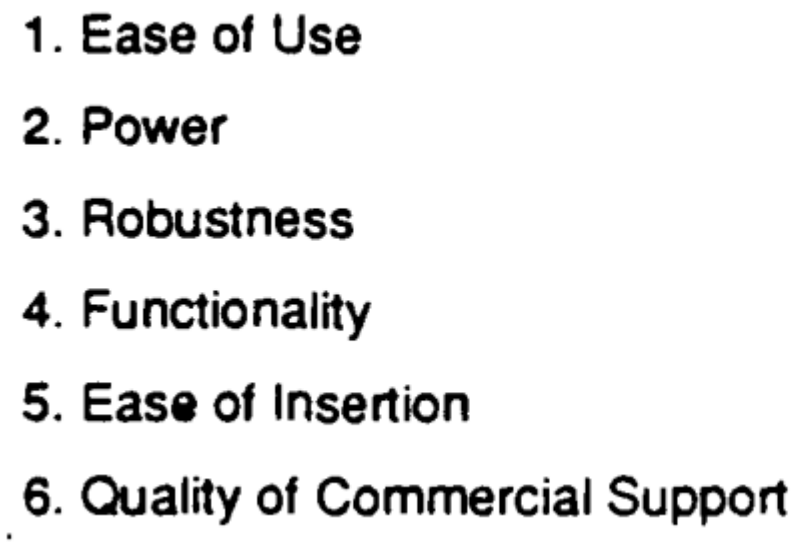
Now, we are going to try to determine a graded classification of software engineering tools. As Figure 6 showed, modern software engineering involves a litany of tools. Wanwisa Roongkaew and Nakornthip Prompoon followed the SWEBOK SDLC phase to derive 10 categories of tools (Roongkaew & Prompoon, 2013):



1) Software requirements tools
2) Software design tools
3) Software construction tools
4) Software testing tools
5) Software maintenance tools
6) Software configuration management tools
7) Software engineering management tools
8) Software engineering process tools
9) Software quality tools
10) Miscellaneous tools issues

Figure 9: Tools Category by Roongkaew and Prompoon

While this reinforces the earlier discussion about the proliferation and pervasiveness of tools across the entire software lifecycle, it will be tedious to re-evaluate the ***software engineering environment*** setup at each phase. We have consulted a report by the Software Engineering Institute (SEI) that takes a different approach in evaluating tools. This approach provides 6 aspects to consider during tools evaluation (Firth, Mosley, Pethia, Roberts, & Wood, 1987):



1. Ease of Use
2. Power
3. Robustness
4. Functionality
5. Ease of Insertion
6. Quality of Commercial Support

Figure 10 : Tools Evaluation Criteria by Software Engineering Institute

Earlier in this paper, we said that the three benefits of a tool are its mechanical advantage, its accuracy advantage, and its consistency advantage. This goes hand in hand with the criteria proposed by the SEI approach. Criteria 1 and 2 fall well under the mechanical advantage while 3, 4 and 6 contribute to accuracy and consistency. Yes, we missed 5, *ease of insertion* (into the environment) which supersedes the other three. In a quest to build a highly productive ***software engineering environment,*** it may seem obvious that we always should strive for all three benefits, but the environment may suggest otherwise.

The SEI report cited above adds that:

<<a tool in and of itself has no value. It is valuable only when applied by a particular individual or organization. Individuals and organizations are different; what is appropriate to one organization or individual may be inappropriate to another organization or individual. >>

We understand from the excerpt above that we should not focus on some metrics of a tool to define how the environment should operate; the environment should dictate the tools to be used. The value of a tool will vary from one software engineering environment to the other. We remind here that a ***software engineering environment*** is the assembly of a team of people producing software with a specified set of tools and processes.

When software engineers want to perform a task that they master, they use a tool just to cope with energy and time savings. They don't expect the tool to dictate how the task should be done, they just want it to follow instructions and work their way. On the other hand, sometimes a black box tool is needed when we want to ensure that everyone performs this task the same way every time. but for now, we retain 3 main environment-based categories of tools:

1. sharp tools: that faithfully follows user-defined execution procedures with no safeguard, yielding a lot of power and flexibility.
2. smart tools: that helps get to the predefined "right" outcome with limited user control.
3. safe tools: that forces that outcome of the task and its execution procedure.

We have already discussed the ***process maturity scale*** and the different phases that could be derived from its observation. We will recall them here as they will serve as graded positions along the process axis in the ***nominal productivity framework.*** The three phases of maturity of a process were the process definition, the process implementation, and the process enforcement. To be complete, we need to add one more phase which represents the absence of processes. We obtain the following grading:

1. undefined processes
2. defined processes
3. implemented processes
4. enforced processes

3.THEORITICAL EXPERIMENTS WITH THE NOMINAL PRODUCTIVITY FRAMEWORK

Now that we have defined the possible input values and the working constraint of our visualization model, we will redraw the framework with labeled input positions:

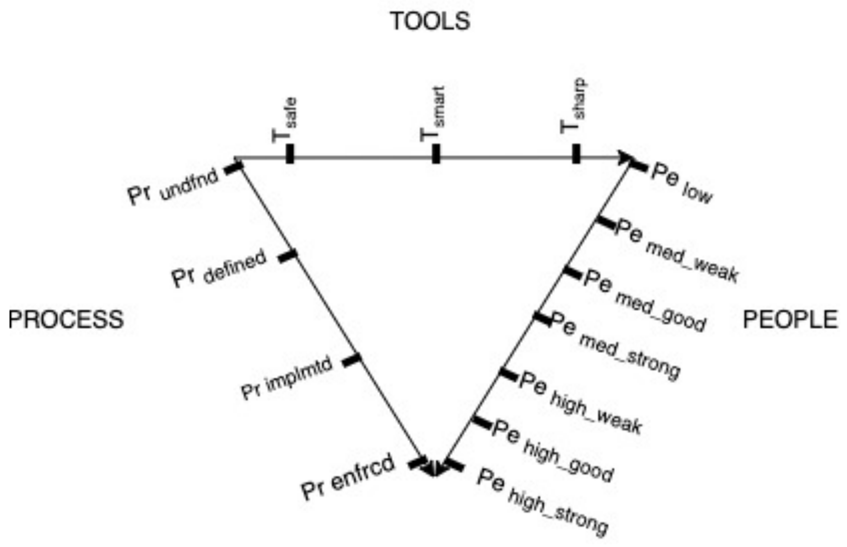


Figure 11 : Nominal Productivity Framework with Labeled Input

The next step is to conduct some theoretical experiments by simulating a ***software engineering environment*** and plotting its coordinates as inputs into the framework and observe the difference in nominal productivity obtained between that environment and another one.

For brevity stake, we have selected 13 simulations that we will draw and analyze in this paper, but the reader is invited to conduct more experimentation of their own by themselves to explore more possibility with the framework.

Later in this thesis, we will call on industry and academia experts to help us understand why this productivity decreases or increases from one environment to the other.

One callout is that the framework can be used at any scope of the ***software engineering environment*** development. We can use this to assess the nominal productivity of the environment for a given task, a set of deliverables in some initiatives, or an entire project. The scoping details will help better position the input along the framework axis and obtain more accurate visualization. For example, just a portion of the team could participate in a particular part of the project so, we would not use the entire team coordinates in assessing the nominal productivity for that portion of the project. We will recalibrate the inputs.

Also, the framework is better utilized when we can propose several simulations and see which one works the best for the case in study. A single simulation by itself does not really give much insight.

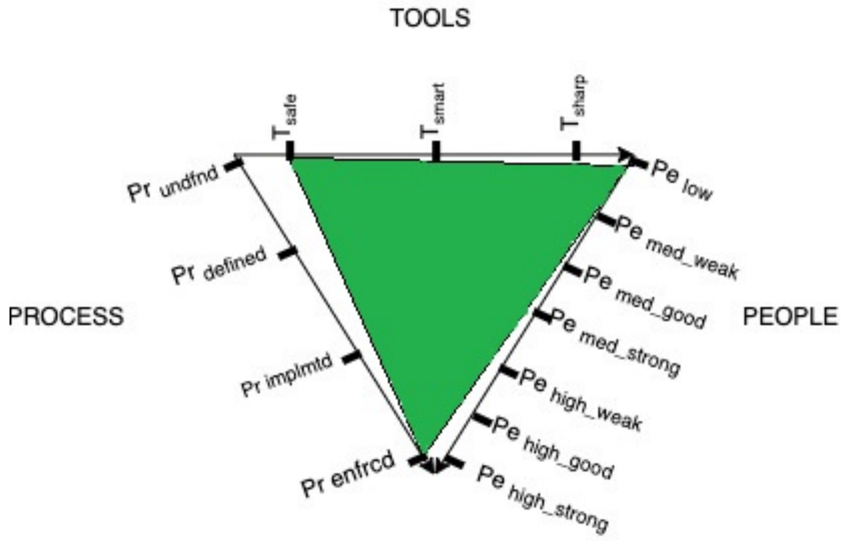


Figure 12: Simulation 1: Software Engineering Environment with low-skilled People, enforced Processes, and safe Tools.

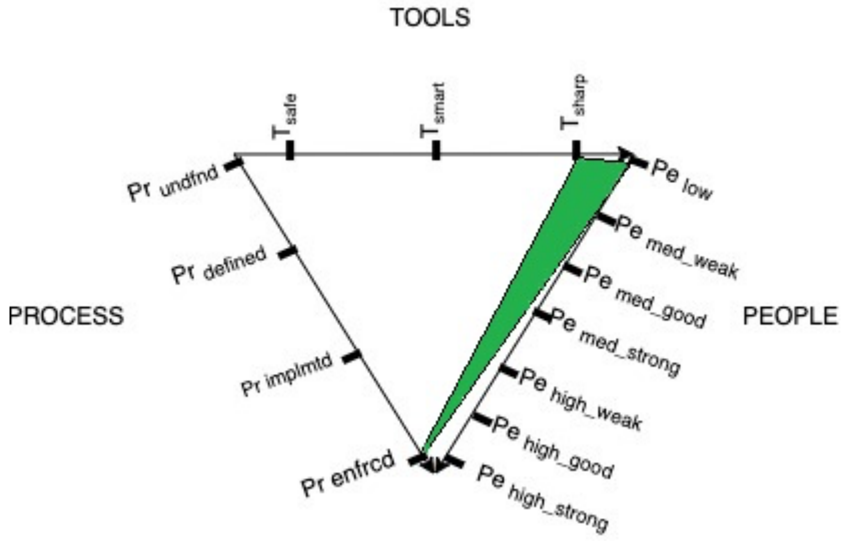


Figure 13: Simulation 2: Software Engineering Environment with low-skilled People, enforced Processes, and sharp Tools.

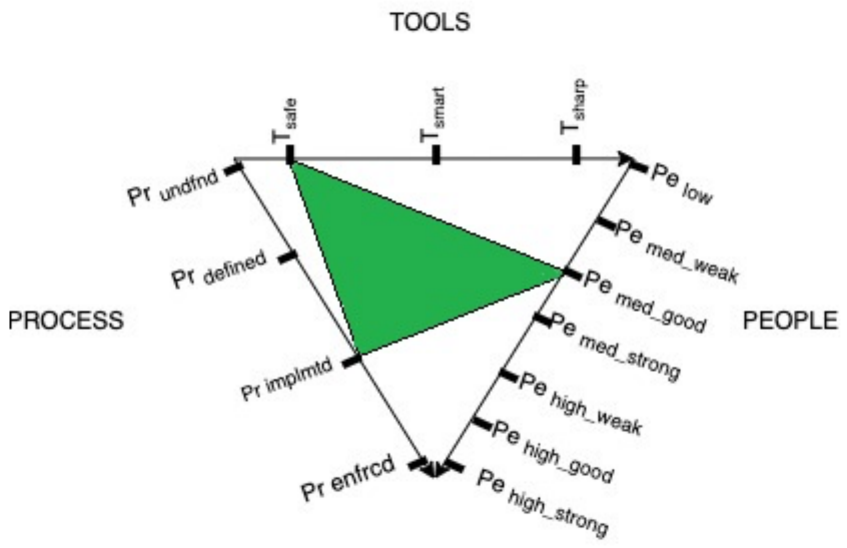


Figure 14: Simulation 3: Software Engineering Environment with medium-skilled and good personality/organization fit People, implemented Processes, and safe Tools.

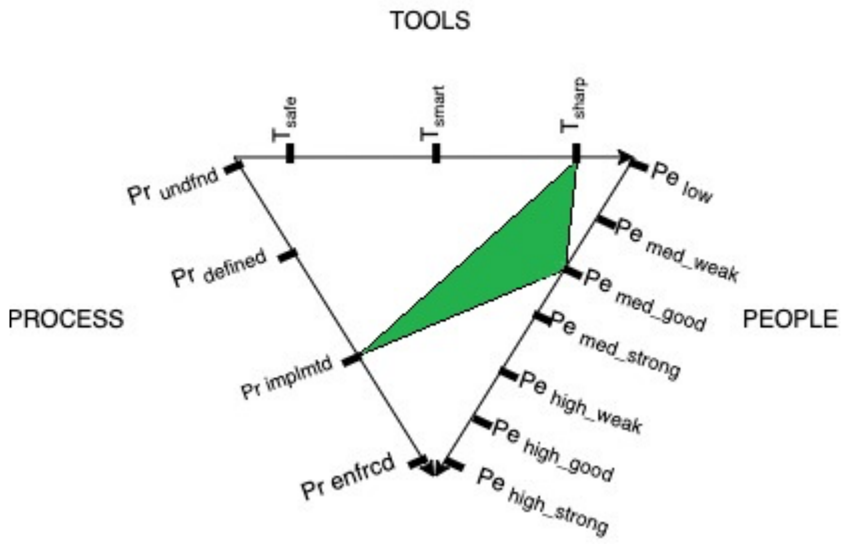


Figure 15: Simulation 4: Software Engineering Environment with medium-skilled and good personality/organization fit People, implemented Processes, and sharp Tools.

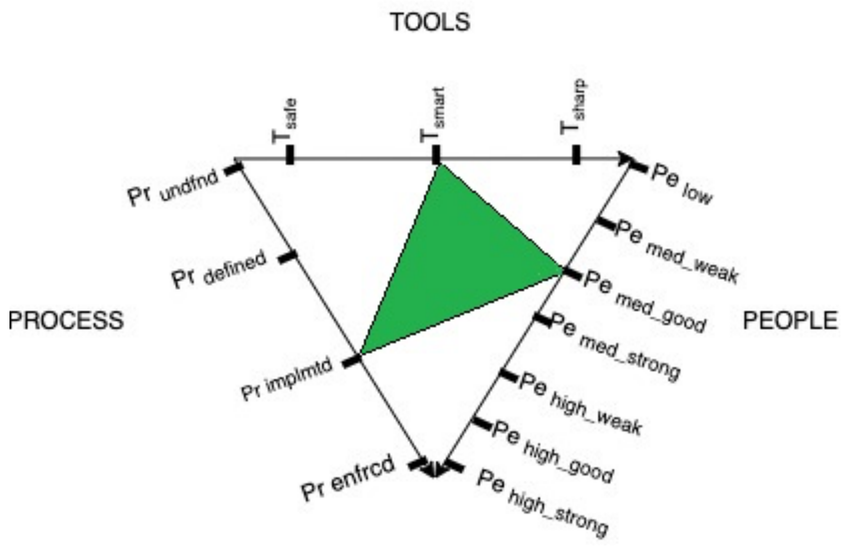


Figure 16: Simulation 5: Software Engineering Environment with medium-skilled and good personality/organization fit People, implemented Processes, and smart Tools.

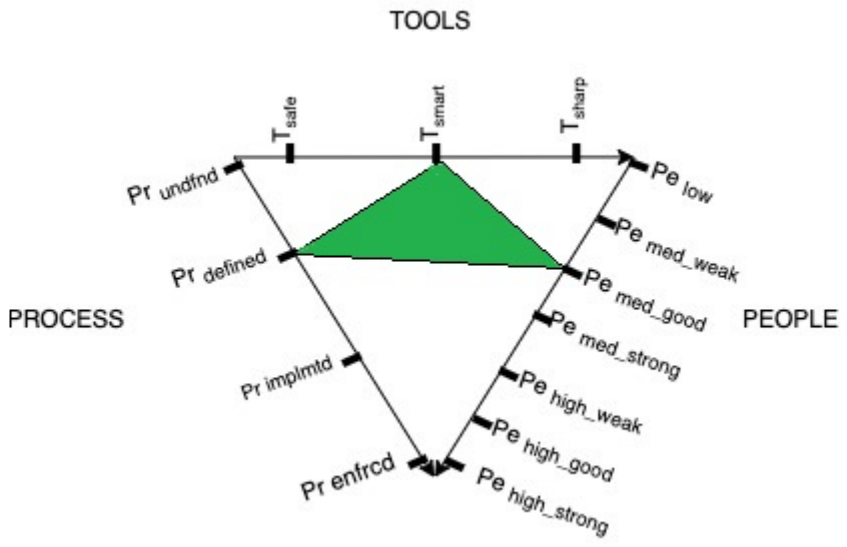


Figure 17: Simulation 6: Software Engineering Environment with medium-skilled good personality/organization fit People, defined Processes, and smart Tools.

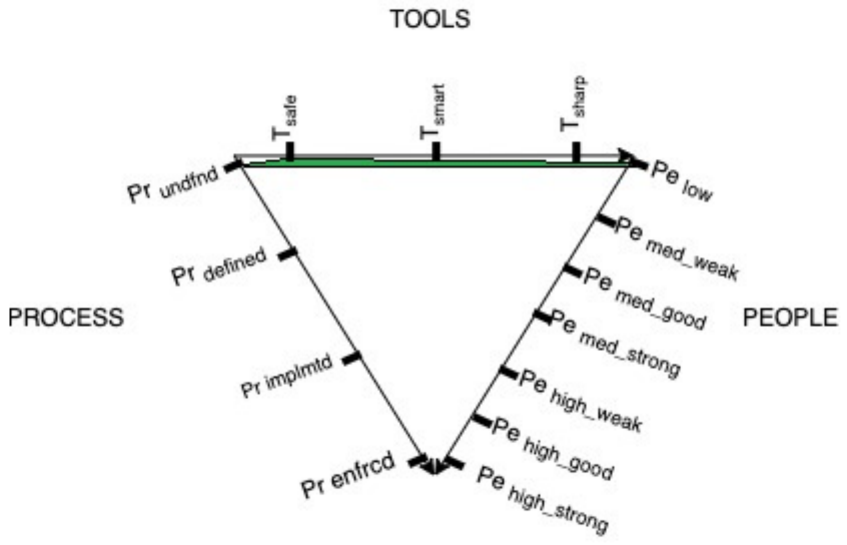


Figure 18: Simulation 7: Software Engineering Environment with low-skilled People, undefined Processes, and safe Tools.

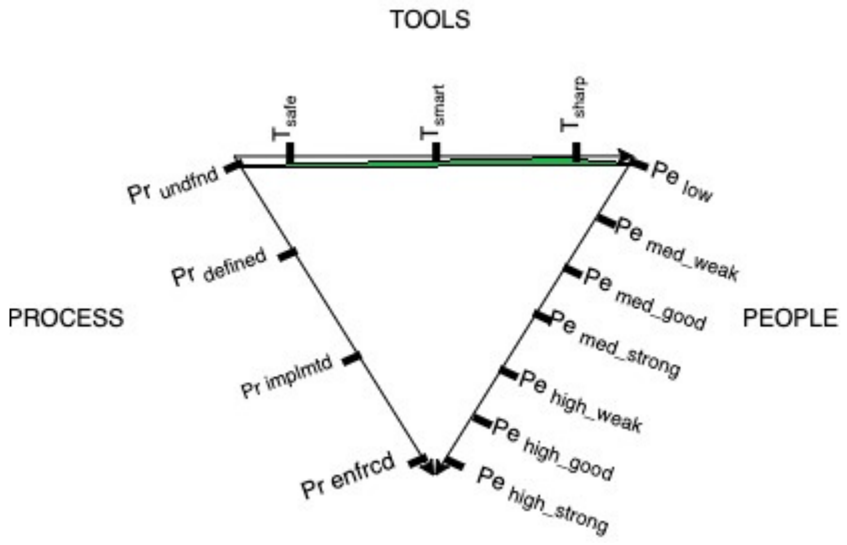


Figure 19: Simulation 8: Software Engineering Environment with low-skilled People, undefined Processes, and sharp Tools.

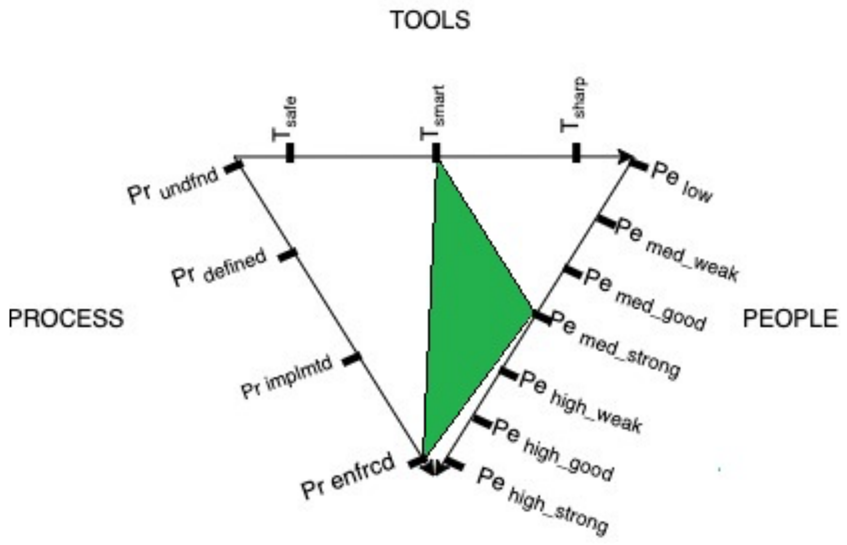


Figure 20: Simulation 9: Software Engineering Environment with medium-skilled and strong personality/organization fit People, enforced Processes, and smart Tools.

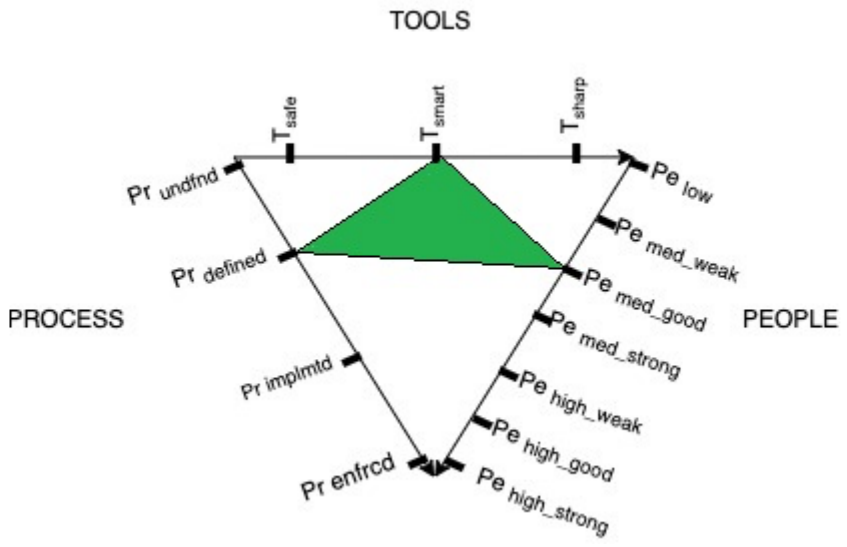


Figure 21: Simulation 10: Software Engineering Environment with medium-skilled and good personality/organization fit People, defined Processes, and smart Tools.

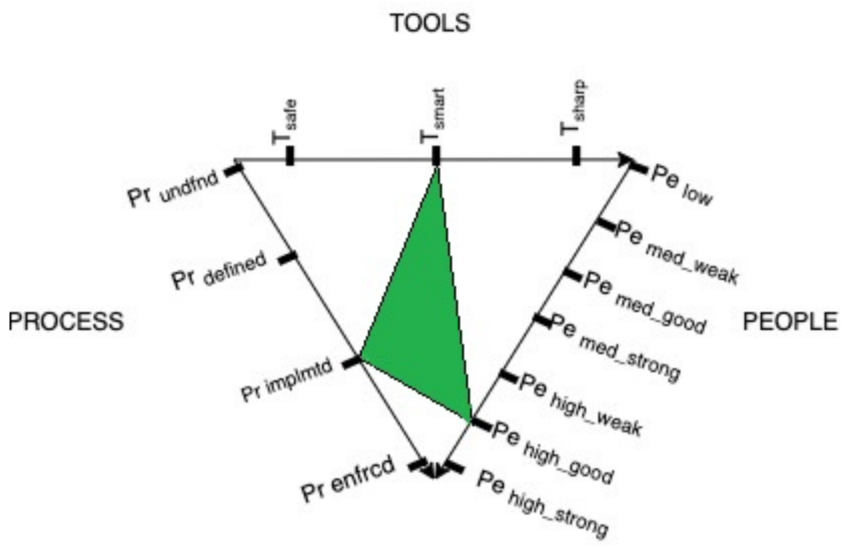


Figure 22: Simulation 11: Software Engineering Environment with highly skilled and good personality/organization fit People, implemented Processes, and smart Tools.

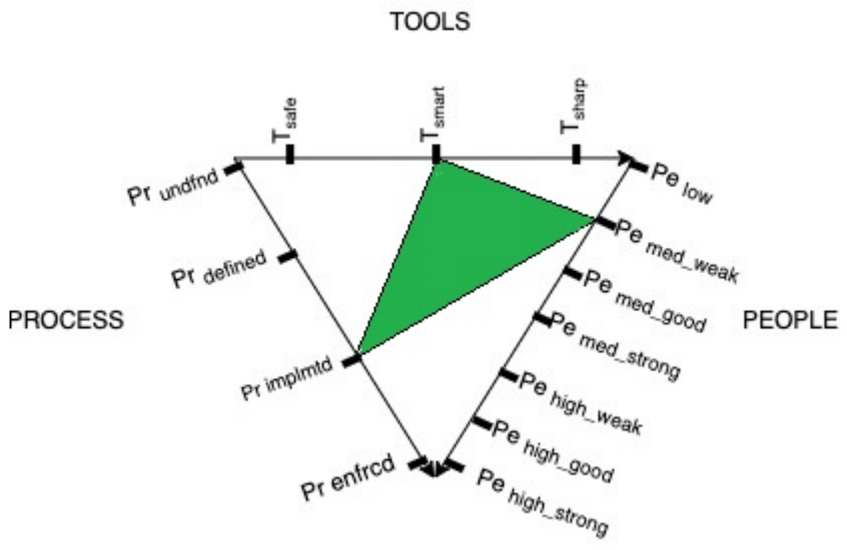


Figure 23: Simulation 12: Software Engineering Environment with medium skilled weak personality/organization fit People, implemented Processes, and smart Tools.

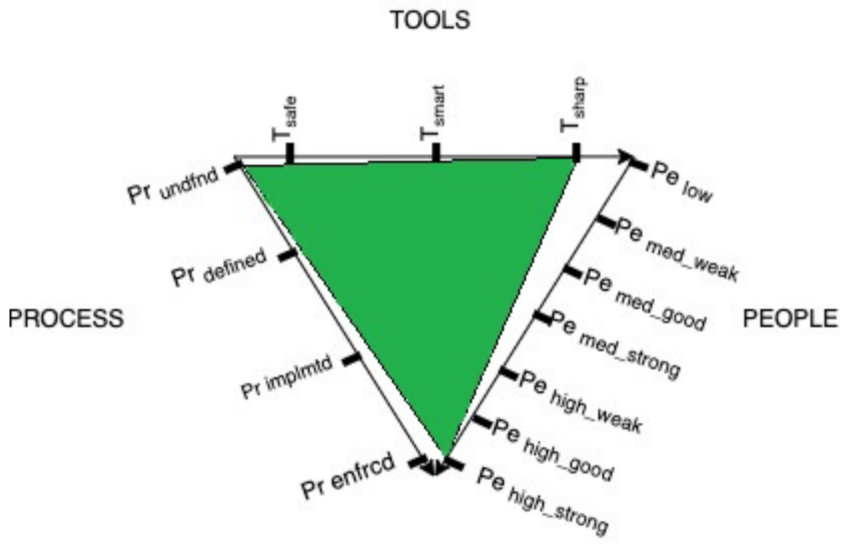
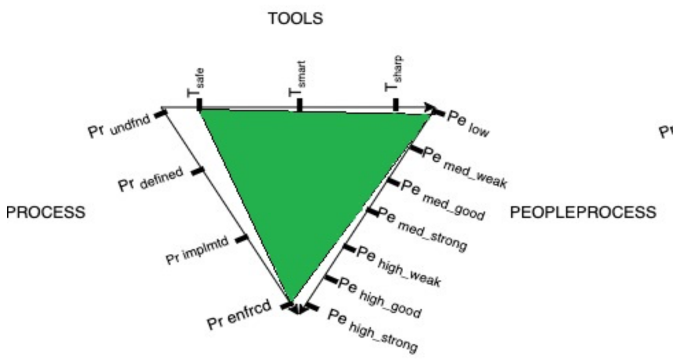


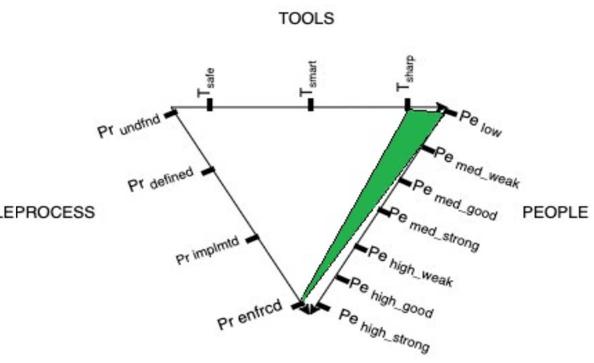
Figure 24: Simulation 13: Software Engineering Environment with highly skilled strong personality/organization fit People, undefined Processes, and sharp Tools.

As we said before, a single simulation by itself does not yield much understanding on how people, processes, and tools affect the ***nominal productivity*** of a *software engineering environment*; we need to analyze two or more simulations side by side to see how the ***nominal productivity*** varies as the states of the inputs change. Let's compare a few environments and attempt to justify the productivity change with the understanding we acquired from the contextual discussion on software engineering productivity intrinsic inputs.

For brevity, we will not analyze all the possible point combinations, but we will emphasize on the key ones to help start the analysis of this framework. Also, since this is an experimental framework, we would like for other researchers to evaluate it with other assumptions and preconditions.

Comparison 1: Env 1 vs Env 2

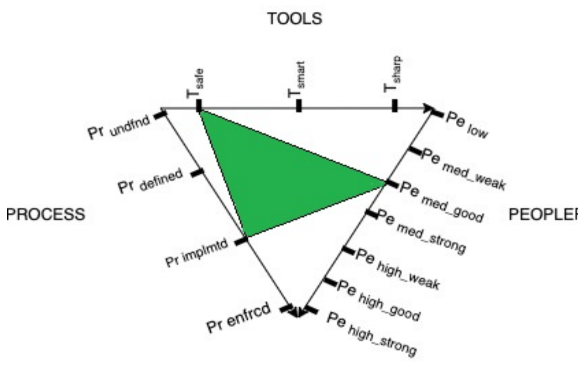


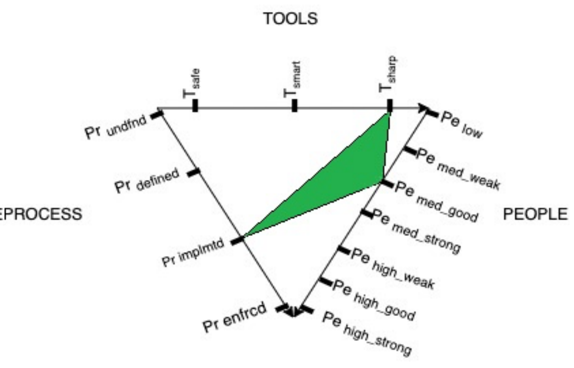


In this scenario, we start with an environment with low-skilled people, enforced processes, and safe tools. We observe that the nominal productivity greatly decreases as we switch from safe tools to sharp tools, everything else being equal.

From the earlier discussion, we recall that safe tools are tools that dictate how a task should be completed and the user is just a trigger for the tools. In this scenario where we have low skilled people it makes sense that the productivity is far greater when we offer them safe tools versus sharp tools which offer more control but require more expertise to do things right.

Comparison 2: Env 3 vs Env 4

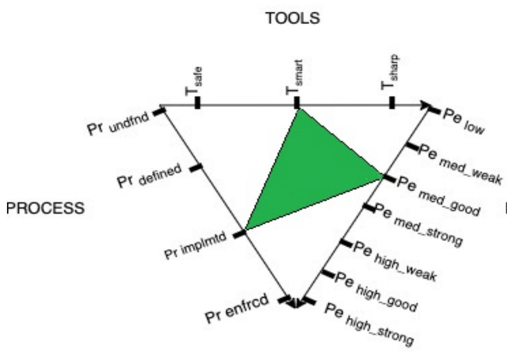


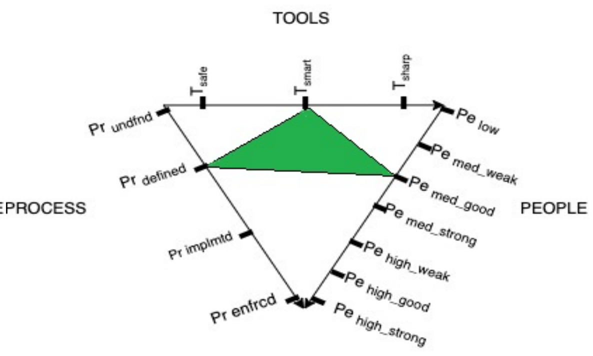


In this scenario, we start with an environment with people with medium skills and good personality/organization fit, implemented processes, and safe tools. We observe that the nominal productivity decreases considerably as we switch from safe tools to sharp tools, everything else being equal.

Once more, we see that when the people's skillset is limited, safe tools are the most suited to achieve greater productivity.

Comparison 3: Env 5 vs Env 6

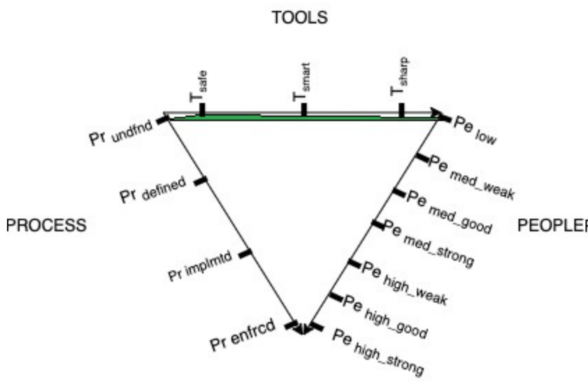


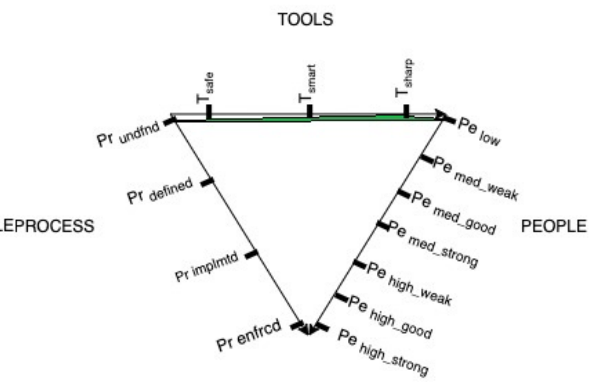


In this scenario, we start with an environment with people with medium skilled and good personality/organization fit, implemented processes, and smart tools. We observe that the nominal productivity does not change significantly as we switch from implemented processes to defined processes, everything else being equal.

As opposed to the two previous experiments where we made big jumps, this time we made a slight variation along one axis. We see that nominal productivity does not respond much to weak stimulus along a single axis. This goes hand in hand with what we explained earlier that productivity is not a continuous function, but it has discrete level like a threshold function.

Comparison 4: Env 7 vs Env 8

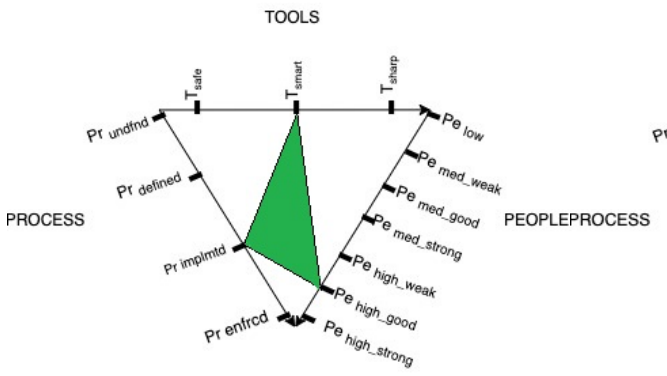


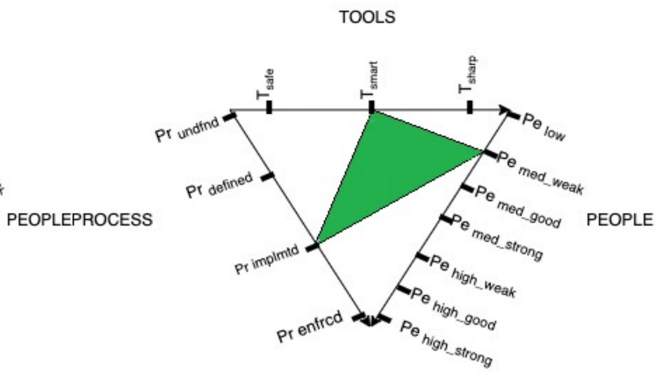


In this scenario, we start with an environment with people with low skills, undefined processes, and sharp tools. We observe that the nominal productivity remains very low as we change sharp tools against safe tools, everything else being equal.

Software engineering is a creative dripline, people with low skillet would typically be less productive, unless a large amount of handholding is invested in terms of both process and tooling.

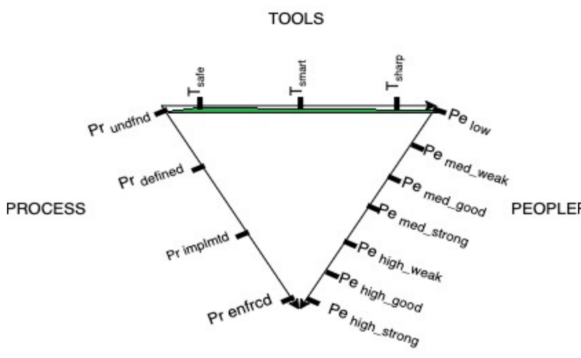
Comparison 5: Env 11 vs Env 12

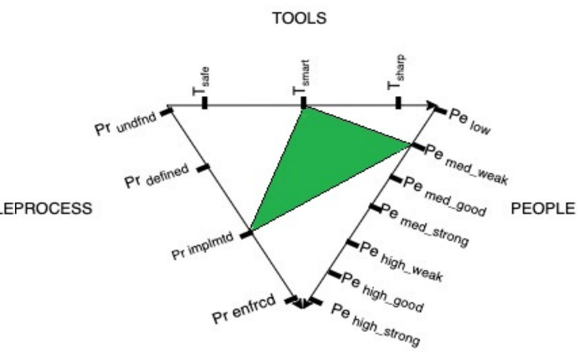




In this scenario, we start with an environment with people with high skills and good personality/organization fit, implemented processes, and smart tools. We observe that nominal productivity does not significantly change as we swap the previous group of people to medium skilled hard personality people, everything else being equal. This is an interesting situation; we would expect a big jump in productivity shifting averagely skilled people to highly skilled people, but the problem is highly skilled people tend to prefer less guidance and perform better in an open environment. The high-level of process and the controlled tooling in this environment imposes the upper bounds for productivity.

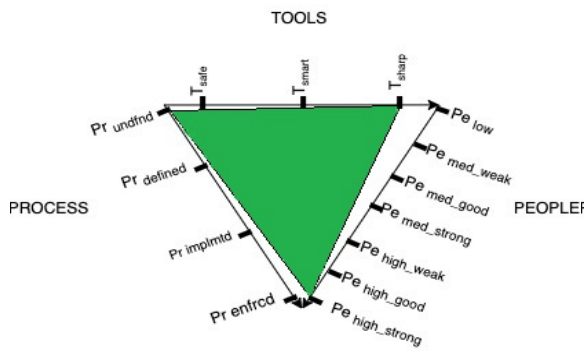
Comparison 6: Env 7 vs Env 12

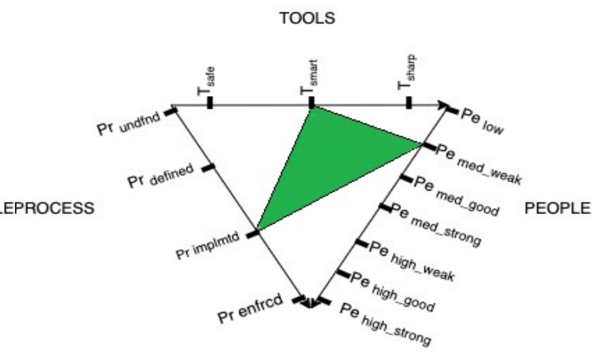




In this scenario, we start with an environment where everything is kept to a minimum (people with low skills, undefined processes and safe tools) and another environment with everything around the average (averagely skilled people, safe tooling, implemented processes). We observe a much greater nominal productivity in the later situation. This again shows how productivity behaves like a does have a threshold function and certain minimum must be met in terms of people, process and tools to realize any significant productivity.

Comparison 8: Env 1 vs Env 12





In this scenario, we compare an optimized environment with highly skilled people, minimal process and sharp tools against another environment aiming to keep everything around the average (averagely skilled people, safe tooling, implemented processes). We observe a much greater nominal productivity in the first situation.
This suggests that if the appropriate tunings are made to create the optimal environment, the productivity is significantly greater than in the average environment.

For brevity, we will not analyze all the possible combinations, but we emphasized some key ones to help start the evaluation of this framework. Also, since this is an experimental framework, we would like for other researchers to evaluate it with other assumptions and preconditions. Nonetheless, we have conducted research involving this framework in representing productivity of which findings will be discussed in the next sessions.

RESEARCH FINDINGS

1. WHAT ARE THE INTRINSIC FACTORS OF PRODUCTIVITY?

We have conducted research with experienced software engineering practitioners to determine the intrinsic factors of productivity. Here is what they say:

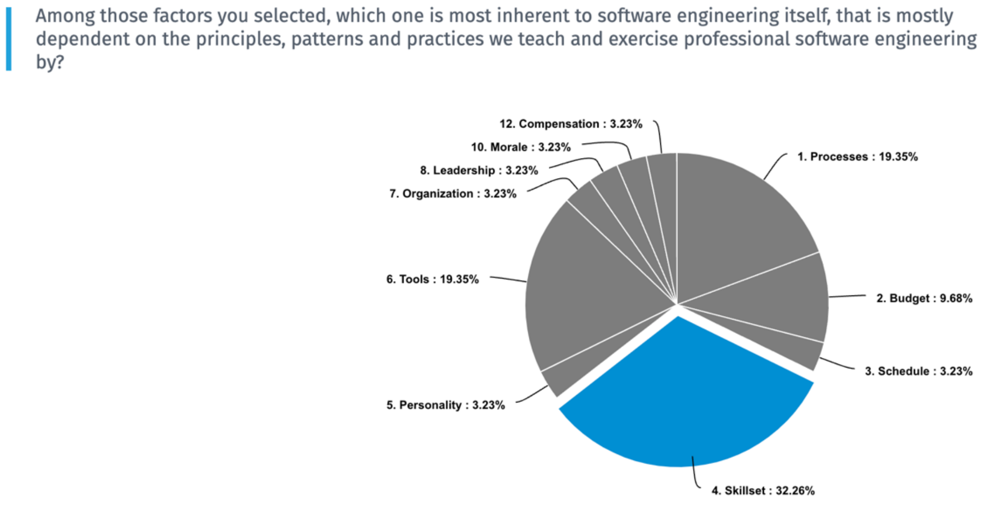

Among those factors you selected, which one is most inherent to software engineering itself, that is mostly dependent on the principles, patterns and practices we teach and exercise professional software engineering by?

Figure 25 : Contributions Weight to Productivity Across Factors

We mentioned earlier in the discussion that skillset, personality, and organization are attributes of people in context of software engineering. We can tally the weight of all those three factors under people, which amounts to 38.72%.

During the coursework for the MSc In Software Engineering, we learned that people, process, and tools are the greatest factors of productivity in software engineering.

We see from the research results that those are also deemed as intrinsic factors by experienced practitioners; these three factors are the only one scoring in double digits with a combined weight of **77.42%.**

We conclude that the research results corroborate the idea that people, tools and process are the intrinsic factors of software engineering productivity.

2. HOW DO PRACTITIONERS SEE INTERACTIONS BETWEEN PEOPLE, PROCESSES, AND TOOLS AFFECT PRODUCTIVITY IN SOFTWARE ENGINEERING?

Alongside our first research question, we also wanted to understand how we can represent productivity as a model of its intrinsic factors. In this paper, we introduced the ***nominal productivity framework*** to help represent productivity as a function of the interactions between people, process and tools states in each ***software engineering environment.*** Other models could help best represent this function, but we took the opportunity in this research to evaluate our own framework to learn more from experienced practitioners.

For this evaluation to be objective, we needed to ensure first that the scenario we have simulated in the theoretical experiments are valid for an evaluation in real-world situations.

We described some scenarios to experienced practitioners and asked them whether those matched either something they have witnessed from their experience or would expect from the expertise in the field.

We summarized the findings in the table below, but a full version of the research questionnaire and answers are available at https://www.questionpro.com/t/7BqvC1Z06xX (all numbers in the following are in percentage of answer from the population).

Scenario#	Scenario Description	Does Scenario Matches Your Expectations?	Does Scenario Matches Your Experience?	Scenario Retained or Discarded from framework evaluation
1	In this scenario, we start with an environment with low-skilled people, enforced processes, and safe tools. We observe that the nominal productivity greatly decreases as we switch from safe tools to sharp tools, everything else being equal.	**73.34** Agree **16.67** Neutral **10.00** Disagree	**23.34** Often/Always **43.33** Sometimes **33.34** Rarely/Never	From these results, the scenario is plausible, therefore **retained**
2	In this scenario, we start with an environment with people			

	with average skills and good personality/organization fit, implemented processes, and safe tools. We observe that the nominal productivity decreases considerably as we switch from safe tools to sharp tools, everything else being equal.	**56.67** Agree **23.33** Neutral **20.00** Disagree	**33.33** Often/Always **43.33** Sometimes **23.33** Rarely/Never	From these results, the scenario is plausible, therefore **retained.**
3	In this scenario, we start with an environment with people with low skills, undefined processes, and sharp tools. We observe that the nominal productivity remains very low as we change sharp tools against safe tools, everything else being equal.	**60.00** Agree **13.33** Neutral **26.67** Disagree	**33.33** Often/Always **40.00** Sometimes **26.67** Rarely/Never	From these results, the scenario is plausible, therefore **retained**
4	In this scenario, we start with an environment with people with average skills and strong personality/organization fit, enforced processes, and smart	**70.00** Agree	**40.00** Often/Always	From these results, the scenario is

	tools. We observe that nominal productivity of this environment is close to that of an environment with people with average skills and good personality/organization fit, defined processes, and smart tools.	**23.33** Neutral **6.67** Disagree	**50.00** Sometimes **10.00** Rarely/Never	plausible, therefore **retained.**
5	In this scenario, we start with an environment with people with high skills and good personality/organization fit, implemented processes, and smart tools. We observe that nominal productivity does not significantly change as we swap the previous group of people to average skilled hard personality people, everything else being equal.	**60.00** Agree **26.67** Neutral **13.33** Disagree	**43.33** Often/Always **23.33** Sometimes **23.33** Rarely/Never	
6	In this scenario, we start with an environment with people			

	with low skills, undefined processes and safe tools. We compare it with an environment with people with average skills and weak. personality/organization fit, implemented processes and smart tools (this environment is aiming to keep everything around the average). We observe a much greater nominal productivity in the later situation.	**56.67** Agree **30.00** Neutral **13.33** Disagree	**33.34** Often/Always **50.00** Sometimes **16.66** Rarely/Never	From these results, the scenario is plausible, therefore **retained.**
7	In this scenario, we start with an environment with people with high skills and strong personality/organization fit, undefined processes and sharp tools. We compare it with an environment with people with average skills and weak personality/organization fit, implemented processes and smart tools (this environment is aiming to keep everything	**70.00**Agree **30.00** Neutral **10.00** Disagree	**43.34** Often/Always **46.67** Sometimes **10.00** Rarely/Never	From these results, the scenario is plausible, therefore **retained.**

	around the average). We observe a much greater nominal productivity in the first situation.			

Table 1: Scenarios Description Comparisons Against Experience Practitioners' Experience or Expectations

After validating the scenario with experienced practitioners, we asked them how effective the ***nominal productivity framework*** was at explaining the change in productivity described in each scenario. The table below captures the essence of their answers.

Scenario#	Scenario Description	Effective	Neutral	Not Effective
1	In this scenario, we start with an environment with low-skilled people, enforced processes, and safe tools. We observe that the nominal productivity greatly decreases as we switch from safe tools to sharp tools, everything else being equal.	66.67	20.00	13.33
2	In this scenario, we start with an environment with people with average skills and good personality/organization fit,	53.34	26.67	20.00

	implemented processes, and safe tools. We observe that the nominal productivity decreases considerably as we switch from safe tools to sharp tools, everything else being equal.			
3	In this scenario, we start with an environment with people with low skills, undefined processes, and sharp tools. We observe that the nominal productivity remains very low as we change sharp tools against safe tools, everything else being equal.	50.00	26.67	23.33
4	In this scenario, we start with an environment with people with average skills and strong personality/organization fit, enforced processes, and smart tools. We observe that nominal productivity of this environment is close to that of	60.00	30.00	10.00

	an environment with people with average skills and good personality/organization fit, defined processes, and smart tools.			
5	In this scenario, we start with an environment with people with high skills and good personality/organization fit, implemented processes, and smart tools. We observe that nominal productivity does not significantly change as we swap the previous group of people to average skilled hard personality people, everything else being equal.	63.33	26.67	10.00
6	In this scenario, we start with an environment with people with low skills, undefined processes and safe tools. We compare it with an	63.34	26.67	10.00

	environment with people with average skills and weak personality/organization fit, implemented processes and smart tools (this environment is aiming to keep everything around the average). We observe a much greater nominal productivity in the later situation.			
7	In this scenario, we start with an environment with people with high skills and strong personality/organization fit, undefined processes and sharp tools. We compare it with an environment with people with average skills and weak personality/organization fit, implemented processes and smart tools (this environment is aiming to keep everything around the average). We observe a much greater	66.67	16.67	16.66

	nominal productivity in the first situation.			

Table 2: Effectiveness of Nominal Productivity Framework at Explaining Scenarios

We see that the effectiveness of the model was rated at around 60%. While this is not a concluding number, it shows that we have taken a step in the right direction. More analysis and refinement will need to be done to the model to improve its effectiveness.

CONCLUSION

In this thesis, we have discussed the factors of productivity in software engineering. We have seen that some factors carry more weight in the way that they affect productivity. An enriching discussion has been presented to help the reader understand those heavy weights factors which we call intrinsic factors in context of the software engineering discipline.

From the master coursework at Kennesaw State University, we learned that the intrinsic factors of productivity are people, processes, and tools.

In this paper, we introduced the notion of a ***software engineering environment*** which captures the states of people, process and tools in given software development organization. We also introduced the concept of ***nominal productivity***, which is the ***open-circuit voltage*** of software engineering team. That is the productivity potential that would have been attained if there were no resistance factors and zero risks during the development of a given piece of software. The goal of the thesis was to validate the assumption we learned in school with experienced industry practitioners as to what are the intrinsic factor of productivity. We also wanted to explore the possibility of mathematically sensible that expresses production in functions of those factors.

After the research, we found that as it relates to people, process and tools being the intrinsic factors of productivity in software engineering, our assumptions were supported by experienced practitioners. We also presented them with our first preliminary version of nominal productivity visualization model that helps explain productivity shift in function of the interactions between states in people, processes and tools. We have studied and analyses this model ourselves in this paper, and even ran some theoretical experiments. We shared those experiments with experienced practitioners and asked them for their feedback on how the model was effective in explaining the scenario described in the experiment. The model was deemed effective at 60%.

In closing, we are proud of the effort invested in sharing our knowledge and discovering more about software engineering productivity. We encourage other students to continue this research on the nominal productivity framework so we can better understand productivity to maximize it.

Bibliography

Academic Accelerator. (Mar, 2023). *extraversion-and-introversion.* Retrieved from https://academic-accelerator.com: https://academic-accelerator.com/encyclopedia/extraversion-and-introversion

Andreessen, M. (2011). Why Software Is Eating The World. *Wall Street Journal*, 1.

Brooks, F. P. (1982). *The Mythical man-month : essays on software engineering.* Boston: Addison-Wesley Pub. Co.

Bureau of Labor Statistics, U.S. Department of Labor. (2021, Aug 3). *Occupational Outlook Handbook, Software Developers, Quality Assurance Analysts, and Testers*. Retrieved from bls.org: https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm

Cain, S. (2013). *Quiet: The Power of Introverts in a World That Can't Stop Talking.* New York: Crown.

Dent, F., & Brent, M. (2015). *The Leader's Guide to Coaching and Mentoring.* Upper Saddle River: FT Publishing International.

Firth, R., Mosley, V., Pethia, R., Roberts, L., & Wood, W. (1987). *A Guide to the Classification and Assessment of Software Engineering Tools.* CMU SEI.

Gallup. (2013). *State of the American Workplace: Employee Engagement Insights for U.S. Business Leaders* . Gallup.

Godfrey, S. C. (1924). The Human Factor in Engineering. *Society of American Military Engineers*, 180-183.

Goldberg, L. R. (1990). An alternative "description of personality": The Big-Five factor structure. *Journal of Personality and Social Psychology*, 1216-1229.

Loten, A. (2019). America's Got Talent, Just Not Enough in IT. *The Wall Street Journal*, 1.

Maese, E., & Robison, J. (2021, Apr 2021). *Measuring Productivity Is Less Important Than Managing It*. Retrieved from Gallup: https://www.gallup.com/workplace/348713/measuring-productivity-less-important-managing.aspx

Mantei, M. (1981). The effect of programming team structures on programming tasks. *Communications of the ACM*, 106–113.

Moritz, T. B. (2021). The 2021 Software Developer Shortage Is Coming. *ACM Journal*, 39-41.

Orosz, G. (2022, Sep). *Can You Really Measure Individual Developer Productivity? - Ask the EM*. Retrieved from The Pragmatic Engineer Blog: https://blog.pragmaticengineer.com/can-you-measure-developer-productivity/

Pournaghshband, Hassan (2021). Effective Process Creation Guide.

In Pournaghshband, Hassan (Ed), *SWE 6633: Software Project Planning & Management*. Kennesaw State University

Roongkaew, W., & Prompoon, N. (2013). Software engineering tools classification based on SWEBOK taxonomy and software profile. *2013 Second International Conference on Informatics & Applications (ICIA).* Lodz: IEEE.

Sackman, H., Erikson, W. J., & Grant, E. E. (1968). Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM*, pp 3–11.

Urciuoli, B. (2008). Skills and Selves in the New Workplace. *American Ethnologist*, 211-228.