Study of Task Processes for Improving Programmer Productivity

by

Damodaram Kamma

Under the Supervision of **Prof. Pankaj Jalote**

Indraprastha Institute of Information Technology Delhi
March, 2017

Study of Task Processes for Improving Programmer Productivity

by

Damodaram Kamma

Submitted
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

to the

Indraprastha Institute of Information Technology Delhi
March, 2017

Certificate

This is to certify that the thesis titled "*Study of Task Processes for Improving Programmer Productivity*" being submitted by *Damodaram Kamma* to the Indraprastha Institute of Information Technology Delhi, for the award of the degree of Doctor of Philosophy, is an original research work carried out by him/her under my supervision. In my opinion, the thesis has reached the standards fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree/diploma.

March, 2017
Prof. Pankaj Jalote

Indraprastha Institute of Information Technology Delhi
New Delhi 110 020

Acknowledgements

Many people walk in our lives, and we learn at least one positive thing from each of them. But a few people create a very strong positive impact on us when compared to others. I have no doubts in my mind that my supervisor Prof. Pankaj Jalote is the one towering personality in my life who had that strong and positive impact on me. I always surprise how he handles so many responsibilities and is able to do justice to each of them. He is the director at IIITD and deals with academic, non-academic, board, private, public and government people, etc. He guides Ph.D. students and also teaches influential academic courses in the Institute. I always wondered if there was any time when he was not available to me for a discussion? Did he ever tell me during the course of my research that he was busy and asked me to discuss later? Never! I did not even witness such a single incident. In fact, he was even ready for a discussion even late in the nights. I also have no doubts that he is one of the nicest human beings I have ever come across in my life. I would like to express my sincere gratitude to him. Each of my discussions with him had stimulated another angle of my understanding and contributed to my knowledge growth.

I would also like to thank my annual review committee member Dr. Rahul Purandare for his continuous feedback that helped me to improvise my research. I would also like to thank my fellow research students Ms.Monika and Ms.Ayushi who reviewed my documents. Let me also take this opportunity to thank non-academic staff Mr. Vinod and Ms. Priti who helped me on many academic issues in time and made working from Bangalore feasible.

I would like to thank the management of Bosch, particularly, Mr. Srikrishnan and Mr. Naved for sponsoring this research. I would like to express my gratitude to Mr. SriGuha V who helped me to meet the heads of various business units and the members of senior management for research approvals. I would also like to thank Mr. Sasikumar, Senior General Manager, who provided all the support that I required for my research. Further, as organization has a policy of scrutiny of research papers in a three-level hierarchical system before publishing in external conferences, though busy with a lot of organizational work, Mr. Sasikumar reviewed my research papers.

Many thanks to the project managers Mr. Nambi Sundarsamy, Ms. Pooja Maruthi, Ms. Parimala Hebbar, Mr. Lam Quoc Hung, and Ms. Swayamprabha for the support they gave to me. Finally, a very big round of applause to all the programmers who video captured their computer monitors and shared the data in this research.

To...

I would like to dedicate this thesis to my wife SOWJANYA who had supported me to pursue my Ph.D. after the six-year stint in software industry. She also encouraged me to complete my course work at the college in Delhi when she was in her last months of pregnancy in Bangalore. I can never forget my daughter RAYNA's unknown sacrifice who was deprived of the precious fatherly emotions and moments when she was a newborn.

Synopsis

Productivity of a software development organization can be enhanced by improving the software process, using better tools/technology, and enhancing the productivity of programmers. While the overall software process and the tools/technology clearly influence the productivity in a project, most of the effort in a project is spent by programmers executing tasks assigned to them, for a given overall process (and tools/technologies used), so productivity is influenced largely by how efficiently programmers execute various tasks. Much of the work on productivity improvement has focused on introducing a process improvement or a new tool in a project or an organization. Less attention has been given to the behavior or practices of programmers for improving productivity.

We know that some programmers are much more productive as compared to others. This is despite a careful selection of people at the entry level by organizations, receiving similar rigorous training, and having similar experience level. The difference between the productivity of programmers in a project is often 2 to 3 times. That is, the productivity of high-productivity programmers (with similar experience levels and background) is sometimes 2 to 3 times of the productivity of average-productivity programmers. What makes a programmer far more productive than an average programmer is an issue that has not been well studied, particularly for peers, i.e. programmers with similar level of experience.

Various studies have been done to study programmers. One set of studies has looked at various practices of programmers for performing specific tasks, in an effort to better understand what programmers do and where do they spend their time. Some studies have also been done to understand differences between expert programmers and novice ones. Various studies

have been done to study the impact of a new tool or practice on programmer productivity. Personal Software Process (PSP) is an approach that focused on self-improvement of processes programmers use.

In this thesis, we focus on the impact of task processes on programmer productivity. A task process is the process used by a programmer for executing an assigned task. Typically, a programmer who is assigned a task of a few days would execute it incrementally in small steps, each step performing some well-defined activity. How the execution of these steps is organized by a programmer is what we refer to as a task process. In the thesis, we propose a general framework for studying how programmers execute tasks assigned to them, and the differences between practices of average-productivity programmers and their high-productivity peers. We also study the impact of transferring the practices of high-productivity programmers to average-productivity peers.

Task process used by one programmer may vary from another for the same task as the overall software process does not standardize any task process. The task processes used by programmers for performing the tasks assigned to them in a project are likely to have an impact on the programmer's productivity. With a better understanding of the relationship between task processes and programmer productivity, it may be possible to improve the productivity of programmers by improving their task processes. In particular, it may be possible to improve the productivity of average-productivity programmers by training them to follow the task process of high-productivity programmers.

In this thesis, we studied the task processes of high- and average-productivity programmers to investigate the following research questions:

- Does a programmer use similar task process while executing a similar type of task?

- Are the task processes of high- and average-productivity programmers similar? And how do they differ from each other?
- Whether the productivity of average-productivity programmers increases by transferring the task processes of high-productivity programmers?

For studying the research questions, we took the following approach. We first identified a few similar live model-based testing projects in Robert Bosch Engineering and Business Solutions Limited, a CMMi Level 5 company. In each project, we identified two sets of programmers - high-productivity programmers and average-productivity programmers. We focused on model-based testing tasks. We studied the task processes of programmers in the two groups to identify the similarity between the task processes used by programmers in a group, and differences from the task processes used by the other group. Finally, we transferred the task processes of high-productivity programmers to average-productivity programmers by training the average-productivity programmers on the key steps missing in their process but commonly present in the high-productivity programmers, and studied the impact of this on their productivity.

For grouping programmers into high-productivity and average-productivity programmers, we requested project managers to rate programmers on productivity as high or average. This subjective evaluation by the project manager was confirmed using the actual productivity data of programmers, which was based on the size of assigned software tasks and the total effort spent for executing the tasks.

For studying the task processes of programmers, we requested each programmer to video capture their computer monitor when they execute their assigned tasks. We then analyzed the task videos for analyzing task processes of programmers. To identify task processes, we first identified a set of steps commonly performed by programmers. We identified new steps while analyzing task videos of programmers. After identifying all the steps used by all programmers, the sequence of steps used by a programmer for executing each task was captured.

For studying the differences between task processes, we modeled each task process as a Markov chain. Each step in a task process is a node in a Markov chain, and the transition between two states represents the number of times a programmer moved from one step to another step. We derived a state transition matrix for each task process. State transition matrix gives the probability of transition from one step to another step. We compared the task processes of programmers within each group (high/average productive) and across the other group by comparing the distance between the state transition matrices of task processes - the larger the distance between the task processes, the lesser the similarity between them and vice versa.

Our study shows that while the task processes of high-productivity programmers were similar to each other, task processes of average-productivity programmers vary within the group. The study also shows that task processes of high-productivity programmers were different from the task processes of average-productivity programmers.

For transferring, we identified the important steps that needed to be changed, added or deleted, and then trained the average-productivity programmer(s) on the steps to be added/modified. After transferring the task process to them, we collected and analyzed new task videos of both high- and average-productivity programmers. We found that the productivity of average-productivity programmers increased and the difference in the productivity between high- and average-productivity programmers reduced significantly. We also found that the similarity between the task processes of high- and average-productivity programmers also increased.

Apart from studying the task processes of high- and average-productivity programmers, we have also conducted few other studies for improving the productivity of programmers. A brief description about each of them is given below.

We investigated the effect of productivity of trainers on the productivity of new programmers and the team by conducting a limited study on a live project. Four senior programmers were selected for training new programmers - a two were identified as high-productivity programmers and the other two as average-productivity programmers. Both sets of trainers trained new programmers. After completion of the training and the new programmers working independently for at least six months, the productivity of new programmers was analyzed. We found that the programmers trained by high-productivity programmers were more productive than the programmers trained by average-productivity programmers. We also found that there was a strong similarity between task processes of new programmers and their respective trainers, further strengthening the findings that productivity of the trainers strongly influences the productivity achieved by the new programmers.

One possible reason for productivity being lower than what is possible may be due to Parkinson's law, which states that work expands to fill the time available for its completion. In a software project if more than needed time is given to a programmer, the extra time will not be revealed as "free time" on the programmer's weekly activity reports, but will result in the programmer consuming all the allotted time resulting in loss of productivity. To counter the effect of Parkinson's law that may be there, we applied a two-pronged approach: allocating one-third less time than the estimated effort for a task, and facilitating issue resolution that may impede progress through a short time-boxed daily meeting. We found an improvement of at least 15% in the productivity of the programmers.

Model-based software development promises to increase productivity by generating executable code automatically from design/models thereby elim-

inating the manual coding phase. Its effect on the productivity of maintenance projects involving enhancement tasks is not well researched. We studied the impact of model-based development on the productivity and quality of maintenance tasks. We observed 173 enhancement tasks done using model-based software development, and 156 enhancement tasks using traditional software development, in six live projects over one year. We found that the productivity of enhancement tasks executed using model-based software development was higher by over 10% as compared to traditional software development.

Contents

List of Figures

List of Tables

Chapter 1

Programmer Productivity

Achievement of business objectives is intrinsically related to the productivity of resources. Productivity is a measure of speed at which inputs are converted into required outputs and is represented as a ratio of output to input. The inputs and outputs used in the calculation of "productivity" depend on the type of organization. For example, an organization involved in manufacturing/production process use number of units produced as an output measure and unit time (or cost) as an input measure.

Productivity is always a prime concern of organizations. Due to the close relationship of business goals and productivity, productivity improvement is almost always an ongoing objective of organizations.

In software, productivity has been looked at with two related perspectives - productivity in a project or an organization, and productivity of individuals. For productivity improvement, at the overall or project level, the focus has been on the software process. For productivity at individual programmer level, the focus has often been on tools and training. In this thesis, our focus is on programmer productivity. It should be pointed out that while our focus is on productivity, it is understood that quality is also of prime importance to organizations and that productivity improvement efforts always have to maintain or improve quality.

In this chapter, we briefly discuss the work in software productivity. We will first discuss overall or project level software productivity, and then discuss background work related to programmer productivity. We end the chapter with an overview of the thesis.

1.1 Software Productivity

Principally, software productivity depends on overall software process, tools/technologies, and programmers as illustrated in Figure 1.1 (Figure taken from [1]).

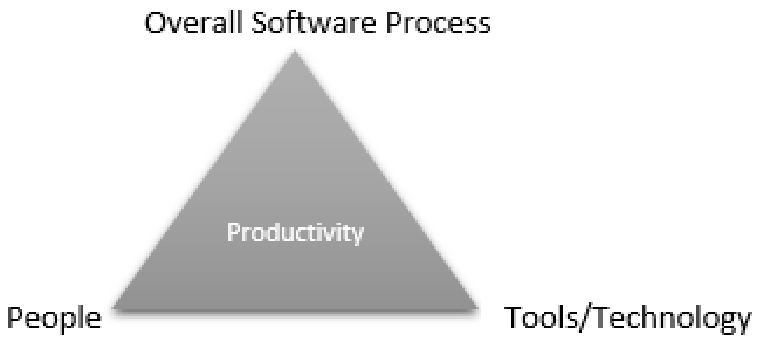


Figure 1.1: Productivity Triangle

Processes for executing software projects have been studied actively for over three decades. Tools and technology to improve productivity has also been an active area of development and improvement and continue to evolve. However, there is insufficient understanding of programmer productivity, particularly at a task level. This thesis focuses on this aspect.

In software, productivity is defined as the ratio of software size to the total effort spent on developing the software [2, 3, 4]. However, even for measuring productivity, there are challenges, both in measuring the size as well as measuring effort.

In this section, we briefly discuss the challenges in measuring size and effort for determining productivity for software. We also briefly discuss what factors have been found to effect software productivity.

1.1.1 Measuring Software Size

The software size measures commonly used in the calculation of software productivity include lines of code (LOC), function points (FP), testable requirements, and use case points. These size measures are further explained as follows:

Lines of code

Lines of code is the most commonly used measure of software size as it is easy to calculate and accurate [5]. However, it has following drawbacks:

- LOC is not available until the end of the coding phase in software development life cycle, that is, LOC can be measured only after the completion of 60% - 70% of overall effort in the entire software development life cycle [6].
- LOC is a poor measure of productivity of an individual, that is, a skilled programmer may write less LOC to develop same functionality compared to a novice programmer [5, 7].
- LOC is obsolete for model-based software development, where programmers do not write even a single line of code but generate code automatically from models that programmers develop [8].
- LOC is a poor measure of size for maintenance projects such as bug fixes [9], and enhancements often require deletion of already existing LOC [10].
- There is no unified acceptance on how to count LOC [11]. Though SEI and IEEE have published some guidelines, the actual practice still needs to be investigated [12, 13]. There are arguments whether comments also need to be counted because comments often help understand the software better and can improve productivity at maintenance phase [14].
- LOC depends on the language used to develop software. Nowadays, a standard measure of LOC is difficult as multiple languages are being used to develop software [8].

Function points

Function points, which is a measure of size, is independent of technology, technique, and programming languages [15, 16]. Computation of FP requires a thorough understanding of system requirements to break functionality into smaller modules that can be easily understood by the system. Functional points are computed based on the parameters like inputs, outputs, inquiries, internal files, and external interfaces of software [17].

Following are the several advantages of FP [15, 16, 17]:

- It can be calculated in the initial phases of software development life cycle (design phase).
- It is a good measure of productivity of an individual as it concentrates on the functionality to be developed.
- It applies to both development and enhancement projects.
- It is also used in model-based software development.

There are also certain drawbacks with FP. Experienced programmers with thorough system knowledge are required to compute FP [18]. Also, the variance of FP measure is more than LOC, that is, each programmer may arrive at a different FP value for measuring the size of same software [19].

Testable requirements

Requirements are predominantly classified into six types: functional, design, interface, performance, implementation, and physical. Testable requirements break each of these requirements into a set of low-level requirements such that each low-level requirement is testable [20]. Further complexity of each low-level requirement is calculated on the basis of the number of inputs it requires, the number of outputs/interfaces it gives, conditions and actions that each low-level requirement calls for, etc. On computing the complexities of the low-level requirements, testable requirements are measured as a function of low-level requirements and their respective complexities. Testable requirements size measure is gaining popularity because it can be computed early in the life cycle of software development and can also be used as a method to analyze requirements and write test cases [20, 21].

Use Case Points

Use Case Points is another size measure computed as a function of actors, scenarios, use cases, etc. It is derived during the requirement analysis phase to capture functionality [22, 23, 24]. Though this measure can be computed early in software development life cycle and is independent of technology, programming languages, etc., it concentrates only on the functional type of requirements and not on other types of requirements. Further, the applicability of this size measure during maintenance phase is also limited [25].

There also exists other size measures like system meter [26], magnitude [27], test-case points [28, 29] etc. However, usage of such size measures is very minimal when compared to the size measures mentioned above. Further, as an effort is spent on various activities, it has been proposed to use a combination of multiple size measures [30, 31]. However, the practice of using a combination of multiple size measures is not as widespread as single size measure.

It is difficult to measure software productivity due to challenges in measuring the size of software. Each size measure has its own advantages and disadvantages. Further, lack of comparable baseline data of software size poses a challenge to analyze productivity improvements.

1.1.2 Measuring Effort

A project manager often estimates the effort required for the software development using either "expert opinion based methods" or "Model-based methods" [32, 33]. He/she also allocates the estimated effort to programmers as a part of scheduling and task allocation process [1]. Programmers, after the allocation, often consume entire effort and may not report “free time” in their weekly activity sheets [34]. Wrong effort estimation complicates the process of measuring correct software productivity. So the computing effort becomes very challenging and is often inaccurate.

Expert opinion-based methods [35] depend mainly on expertise, experience, and analogy of the experts involved. A few of them explore analogous conditions/situations to identify the effort spent on completing similar things in the past [36]. Few others estimate the effort based on their experience and expertise. One of the notable methods includes “Delphi method,” where opinions of more than one expert are considered to compute the required effort [37].

Model-based methods are mainly the regression-/algorithmic-based methods. These methods require first identifying factors that influence the effort and then using regression methods/algorithms to fit a relation between the effort and its factors [30, 31]. One of the notable works includes “Cocomo model [37],” where effort is computed as a function of “lines-of-code.” Some studies also use artificial intelligence/machine learning approaches for computing effort [38]. Most often, these model-based methods are localized to a particular working environment and require recalibrating/fine tuning for other working environments.

Measuring the actual effort spent on tasks in a project by programmers is very challenging. Project manager schedules and allocates tasks (along with estimated effort for executing those tasks) to programmers [1]. Programmers often report to their project manager the effort spent on executing his/her tasks using weekly timesheets or activity reports [39, 40]. However, as programmers often spend effort on many tasks (including non-project activities) in a week [41], it is difficult for a programmer to remember and report back the exact effort spent by him/her on various project related tasks at the end of the week. Further, the manual data collection is often error prone [42].

There exists various time tracking tools to monitor the programmers and capture automatically the effort spent by him/her [43, 44]. Researchers have used such type of tools to study few programmers for a limited time duration [41]. However, usage of such tools in a large software development companies on a larger group of programmers for a longer time period is limited [39, 45]. Further, data collected from automatic tools often will be in the format of event and its time stamp [43]. The data has to be analyzed properly to extract the correct effort spent on project tasks. Additionally, automatic capturing the effort spent by programmers away from their computers is challenging.

There also exists various semi-automated effort capturing tools prompting programmers at regular intervals to enter their effort details [46, 47]. Semi-automated tools impose context switching while programmers are executing their tasks and thereby reducing the productivity [48, 49].

Challenges in the accuracy of measuring effort data spent by programmers further complicates the computation of software productivity.

1.1.3 Factors Affecting Software Productivity

Detailed factors that affect software productivity have been identified on the basis of surveys, interviews, and observation of programmers and project managers. Some of them are:

- *Tom Demarco:* [50] Environment factors like light, quietness, noisy work environments, etc. affect the programmer productivity more than salary, language, years of experience, etc.
- *Brooks:* [51] The complexity of programs has a significant effect on productivity.
- *Rasch:* [52] Ambiguity and conflict in programmer's role can also affect productivity.
- *Black Burn and Scudder:* [53] Project duration and team size have a significant impact on productivity.
- *Macaulay:* [54] Experience, knowledge, motivation, tools, and techniques are important factors to improve productivity.
- *Port and McArthur:* [55] Object oriented programming improves productivity.

- *Kitchenham:* [3] Reuse of software artifacts is a significant factor to improve productivity.
- *Berntsson:* [56] Requirements accuracy, schedule, and customer involvement also have their impact on productivity.
- *Spiegl:* [57] Business culture, promotions, freedom, responsibility, appreciations also affect productivity.
- *Goncalves:* [58] Collaboration and communication between stakeholders has an effect on productivity
- *Bailey:* [49] Task switching's and interruptions have a significant impact on productivity.
- *Ronald:* [59] Goals set to project/programmers also have an impact on productivity
- *Humphrey:* [60, 61] Personal software processes of programmers impact software productivity.
- *McGibbon:* [62] Adoption of process frameworks improves productivity
- *Marcelo:* [63] Work dependencies between teams/programmers impacts productivity. Further, correct team/programmer has to be deployed to match technical dependencies for improving productivity
- *Bruckhaus:* [64] Usage of tools and automation improves software productivity.
- *Thomas Tan:* [65] Attrition of team members in a project affect productivity

Stefan Wagner and Melanie Ruhe classified the factors into technical and soft factors after an intense research on the factors that affect software productivity[66].Technical factors are divided majorly into product-, process- and environment-related factors. Some of the product-related factors include execution time constraints, product quality, user interface, required software reliability, etc. Process-related factors include process maturity, the time span between major changes, project duration, concurrent hardware development, etc. Environment-related factors include the use of software tools, programming language, modern development practices, etc. Soft factors mainly deal with corporate culture, team culture, individual capabilities, experiences, work environment, etc.

1.2 Software Process Improvements for Improving Productivity

During the last three decades, focus was on the importance of the overall process of software development on software productivity. This naturally led to an increase in emphasis on improving software process for improving software productivity [67]. Improvements in the overall software process increase software productivity by identifying and eliminating waste during the software development and optimizing existing methods to reduce the software development effort. Recognizing the challenges in software process improvements, some frameworks have emerged to help organizations improve their process. Some of them are briefly described here.

- ***CMMi:*** Capability Maturity Model Integration (CMMi) is a framework introduced by SEI, CMU to assess and improve overall software process for better productivity [1, 68]. Processes of projects/organizations are evaluated to know the maturity level of an overall software process. Maturity levels, under CMMi framework, are classified into the following five levels: Initial, Repeatable, Defined, Quantitatively Managed, and Optimized [69, 70]. Organizations improving its CMMi maturity level by one have reduced their development effort resulting in productivity improvement [71, 72, 73, 74]. Projects/Organizations certified with level 5 (optimized) rating are considered to have mature overall software processes and strive for continuous quality/productivity improvement [75].

- ***ISO:*** International Organization for Standardization (ISO) determines the process and product capabilities and improvements [76]. Unlike CMMi that concentrates only on software processes, ISO is a generic platform used across many industries for evaluation of both processes and products [77]. Adopting the guidelines and standards of ISO in software projects had improved quality [78] and productivity [79].

- ***ASPICE:*** Software Process Improvement and Capability Determination is a systematic framework for designing and assessing automotive software development processes [80]. It was developed in consensus with Original Equipment Manufacturers (OEM) in the automotive domain. Like CMMi, ASPICE also has maturity levels. However, unlike CMMi and ISO that predominantly deal with project management practices and improvements, ASPICE concentrates more on engineering methods, management and improvements [81]. Many CMMi Level 5 software development companies failed to achieve ASPICE level 5 maturity level [82].

- ***DOI78:*** This framework is mainly used to assess avionics-related software [83]. This framework first classifies the avionics software into one of the five levels (based on the existence of bugs in the software): A, very critical and causes damage to life; B, critical when immediate action is not taken; C creates panic when action is not taken; D to E, non-critical. Further, according to the level of the software, this framework imposes various engineering and project management practices in software development [84].

Though these frameworks lay down the necessity of continuous process improvements, they hardly mention any research methodology to carry out process improvements. Some of the well-defined research frameworks available in the literature include quality improvement paradigm [85], goal question metrics [86], six sigma [87], etc.

Quality Improvement Paradigm

Basili established the procedure for process improvements in organizations through Quality Improvement Paradigm (QIP) [85]. QIP is an iterative process and contains two cycles, namely organizational learning cycle and project learning cycle, as illustrated in Figure 1.2. Project learning cycle is a part of organizational learning cycle and is specific to a project in which improvements are planned or implemented. Further, project learning cycle is iterative in nature. Six steps defined in QIP are as follows:

Characterize and understand: Understand existing processes and environment to identify the areas of improvement.

Set goals: Define quantifiable goals from different perspectives (customer, project, organizational, etc.)

Choose processes, techniques or tools: Choose processes, techniques, methods, and tools such that they work well with current processes and environment to achieve the set goals.

Process executions (also called project learning cycle): Test the chosen processes, methods, and tools in a few projects. This step contains three sub-steps: executing the chosen processes/methods/tools, collecting data from projects for analyzing results, and providing feedback for any corrective actions.

Analyze results: Analyze results (and feedback) from all projects and identify whether or not the goals are achieved. This step also recommends future improvements that are required.

Package and store experience: Store experiences, results, etc. for future reuse.

Goal Question Metrics Approach

There exist a lot of metrics in software development [88]. Collecting the data of all metrics in an experiment/study is costly and time-consuming. However, not collecting the data of required metrics may demand a repetition of the entire study/experiment [89]. Therefore, even before the beginning of actual study/experiment, an optimal number of metrics that are only sufficient to analyze a set goal needs to be prepared.

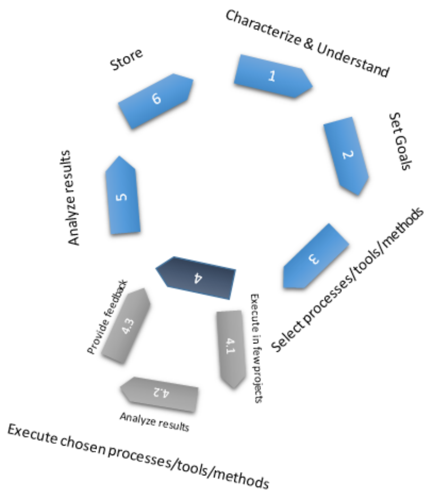


Figure 1.2: Quality Improvement Paradigm

Though QIP gives an overview of the way to carry out process improvements in an organization, it hardly specifies an approach to identify metrics/measures that are required to analyze a set goal. Goal Question Metrics (GQM) [86] addresses this issue through a three-layered hierarchical approach as illustrated in Figure 1.3. GQM requires specifying goals of a study (at the top layer of this hierarchy) in a particular manner comprising purpose, object, issue, and viewpoint.

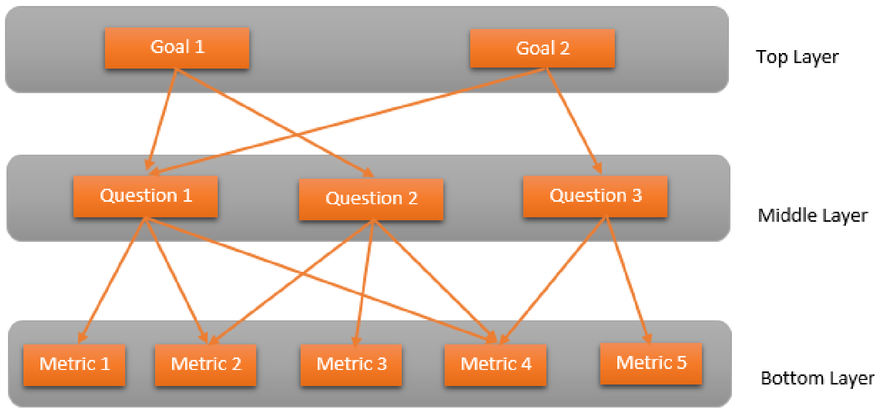


Figure 1.3: Goal Question Metrics

On specifying the goals of a study, each goal is refined into a set of questions (in the middle layer) such that all these questions together solve/achieve/answer a goal.

After deriving various questions that need to be answered to analyze a set goal, each question (in the middle layer) is refined into a set of metrics (at the bottom layer) such that all these metrics are captured and analyzed to study each question.

Six Sigma

Six Sigma defined by GE [87] has the following two key methodologies: DMAIC [90] and DMADV (both inspired by Deming's Plan-Do-Check-Act cycle [91]). DMAIC is used to optimize and bring necessary improvements in the existing processes while DMADV is used to improve specifically the design of products [92, 93]. A brief overview about DMAIC and DMADV is given below [94].

The basic DMAIC methodology consists of the following five steps:

- *Define* process improvement goals.
- *Measure* important characteristics of existing process and collect related data.
- *Analyze* the collected data to verify relationships between them.
- *Improve* or optimize existing process by incorporating changes in the existing process. This step also includes analyzing the collected data to verify improvements.
- *Control* any deviances from goals by bringing necessary changes and continuously monitoring the existing process.

DMADV is mainly used to optimize the product design. The basic methodology consists of the following five steps:

- *Define* design goals.
- *Measure* and identify quality characteristics of the product.
- *Analyze* various design alternatives to choose the best design.

- *Design* a plan for verification of the chosen best design.
- *Verify* the chosen design before handing over the same to process owners.

1.3 Individual Productivity - Work in Other Disciplines

The notion of improving productivity of people started in industrial engineering at the end of nineteenth century. Taylor observed how each worker execute his mechanical tasks. His observations, based on time-motion study, were developed into a theory called Taylorism also called scientific management theory [95]. This theory helped many organizations improve productivity [96]. Some of the principles of Taylorism (scientific management theory) include

- Few individuals are more talented than others, and there is always one best way of performing any task.
- Organizations should discover best ways of performing a task and should train individuals on the same.
- Work has to be divided into sub-tasks, and dedicated skilled people should be engaged on each of the sub-tasks.
- Role-based division of work improves productivity. Work has to be divided between two groups of people - managers and workers. The role of managers is to observe workers, identify best ways of performing tasks, and train workers on the same. The role of workers is to perform tasks assigned by their managers.

Observing how individuals execute their tasks help many organizations discover the best way of performing a task and bring mechanization/automation in work to improve productivity [97].

Scientific management theory is best seen in lean methods used by manufacturing organizations [98]. Lean methods study processes followed (used) by individuals while executing their tasks to identify and eliminate the waste in their process for improving productivity [99].

Standardized work

"Standardized work" is a lean method followed by Toyota for improving productivity and quality in a production process [100]. The main idea behind this method is to observe and document the sequence of steps used by people in a production line for uncovering the best production line process for improving productivity and quality (to the required level). Major steps in this method are as follows:

- Fragmenting a task into major steps.
- Documenting the steps involved in executing a task in a work sheet.
- Recording the time spent on each step in four categories: manual time, automated time, wait time, and walk time.
- Calculating the overall cycle time of the present process.
- Analyzing and modifying the present process to achieve required cycle time.
- Training workers on the new process.
- Standardizing the new process.

Value Stream Mapping

Value stream mapping (VSM) is another lean method used by manufacturing industries to identify and eliminate waste in their process [100]. VSM documents both material and information flow of current process. VSM considers not only the product flow but also the management information system flow that supports the product flow. Each step in the process is mapped into one of the two categories "value added" or "non-value added" for reducing the non-value added steps from existing/current process [101].

Both the standardized work and the VSM observe the work done by individuals to identify best ways of performing a task for improving the productivity of individuals [102].

Scientific Management Theory in Software Engineering

Some of the principles of the scientific management theory are seen already in software development.

- Software development life cycle models split work into different phases - requirements, design, coding, and testing [103].
- People trained specially to work on each of the phases are called requirement analyst, architect/designer, developer, tester, etc.
- Project team contains a manager and team members. The responsibility of project manager is to select team members, train them on required skills, allocate tasks, and ensuring the quality of work within a given project schedule. The responsibility of team members is to execute their tasks assigned by their project manager [104].

The principle of "One Best Way" of scientific management theory in software engineering is currently attributed to selecting best tools, methods/techniques for executing a project and is not directly attributed to how people execute their tasks. Regarding the principle of "one best way," it is worth to mention what Agresti has mentioned about scientific management and software engineering [105].

"I was also impressed that much of what they were saying then about I.E. (or 'scientific management' as it was known then) could be said today about software engineering. Now, among the various methods used , there is always one method which is quicker and better than the rest. And this one best method can only be discovered through a scientific study and analysis of all the methods in use "

One can argue that there is a significant difference between manual and intellectual work; hence, "one best way" may not be applicable to knowledge worker. It is also important to know what Humphrey has mentioned about scientific management in relation to intellectual and manual work [106].

"Even though manual and intellectual tasks are significantly different, we can measure, analyze, and optimize both and thus apply Taylor's principles equally well. The principal difference between manual and intellectual work is that the knowledge worker is essentially autonomous. That is, in addition to deciding how to do tasks, he or she must also decide what tasks to do and the order in which to do them. The manual worker commonly follows a relatively fixed task order, essentially prescribed by the production line. So studying and improving the performance of intellectual work must not only address the most efficient way to do each task but also consider how to select and order these tasks. This is essentially the role of a defined process and a detailed plan. The process defines the tasks, task order, and task measures, while the plan sizes the tasks and defines the task schedule for the job being done."

Although one best way seems to be a setback to innovation, in reality, they complement each other. One best way is evolved from the current innovative practices. Standardization of "one best way" helps further innovation for improving productivity and quality [107].

1.4 Programmer Productivity

We have classified related work on programmers into a few categories: Studies observing work practices of programmers, studies identifying the differences between programmers, and the studies that tried to modify the work practices of programmers for improving productivity.We briefly discuss each of these. We also discuss personal software process related studies, and pair programming as they also impact the personal productivity of programmers, and some other related works.

1.4.1 Studies Observing Work Practices of Programmers

Martin observed five programmers for about two hours while the programmers were investigating and fixing a bug in a given piece of code. The bug fix requires a code change in 5 methods in a particular order. Martin video captured the screens of programmers and manually annotated the videos to identify the sequence of methods analyzed/modified by each programmer. He also captured the navigation techniques from one method to another and classified them into five categories (Scrolling, Browsing, cross-reference, recall and keyword search). He observed that one programmer had finished the task altogether. He found that the programmers who did not finish their task tried to modify the entire code in only one or two methods. Further, based on the frequency of methods revisited by programmers, Martin found that Programmers who did not finish their task tried to investigate the code using an opportunistic approach but not a systematic approach. Additionally, the programmer who had finished the task used "keyword and cross-reference search" for moving from one method to other method indicating that program comprehension is also different between programmers [108].

In another study, Robert W. Bowdidge observed six pairs of programmers (from academia and industry) while the programmers were restructuring a given piece of code for about two hours. He video captured the computer monitor of each pair and also audio-taped their conversation. Further, He manually analyzed the audio/video tapes and found that two pairs of programmers followed a well-planned systematic approach while remaining four pairs of programmers mostly used an exploratory approach. The tools selected by various pairs of programmers for restructuring the code seem to be different. Further, the features of the tools selected for code restructuring influenced the execution behavior of task [109].

Joseph Lawrence video captured the computer monitors of 12 programmers while the programmers were fixing their assigned bugs. He asked programmers to think aloud while they were working. Each programmer was observed for 2 hours. The audio tapes of programmers and videos of their computer monitor's were analyzed and mapped them into seven activity codes (hypothesis start, hypothesis modify, hypothesis confirm, hypothesis-abandon, scent to seek, scent gained and scent lost). They found that programmers predominantly require scent (information) for hypothesis (understanding and fixing bugs). Therefore, he suggested tool designers include various ways of providing scent to the programmers while designing a tool [110].

Roman Bednarik and Markku Tukiainen used an eye tracking device in their experiment to understand how programmers comprehend a program. He observed the eye movement data of 18 participants and found that programmers exhibit a different behavior from others. Few programmers tried to run through the code whereas few others concentrated on areas of visualization [111].

Jason Singer conducted a web-based survey of 13 software engineers (out of which six were responded) and found that programmers spend time on various activities like documentation, meetings, etc. along with coding. Later, Jason manually observed programmers for half an hour once in every three weeks for about six months and found that programmers perform various activities like search, documentation, coding, debugging and modifying the code. Further, searching is the most frequently used activity by developers [112].

In a recent study, AN Meyer first did a web-based survey of 379 software professionals and then manually observed 11 software developers for about 4 hours (- 2hours before lunch and two hours after lunch) and found that programmers spend time on both work and non-work related activities. Further, task and activity switches are commonly observed. Additionally, programmers perceive themselves as high productive when they have least task/activity switches [113]. This study was corroborated well with the author's another study where they observed the programmers using a monitoring tool [41].

Van Solingen conducted a web-based survey and found that interruptions cause task/activity switches. Further, these interruptions happen majorly due to other persons, emails, and phones. Programmers can avoid interruptions due to emails and phones, but it is often difficult to avoid interruptions arising due to direct interactions of other persons [114].

Czerwinski asked 11 programmers to record manually their activities and analyzed those dairies to identify reasons for switching between activities. They found that 40%

of task/activity switches are self-initiated followed by 19% due to new/next tasks. Few other causes include 14% due to phones, 9% due to meetings, 3% due to emails and 1% due to other persons [115].

Andrew observed 17 developers and manually recorded their activities in 90-minute sessions and found that developers sought almost 21 different types of information for executing their tasks like understanding execution behavior of code, reasoning about design, etc,. Further, programmers have to switch tasks/activities when the source of information is with unavailable coworkers [116].

Robert Minelli recorded various events performed by 18 developers in an IDE and found that developers spend time on various activities like editing, navigation, inspection, debugging, etc. Further, Program comprehension is the primary activity that effects the productivity of programmers [117]. Sanchez also studied the interaction logs of developers with IDE and found that the productivity of programmers degrades based on the prolonged duration of interruptions [118].

Goncalves manually observed 14 programmers for 4 hours and found that each programmer spends around 40% of the time on individual activities and 45% of the time on collaborative activities interacting with other programmers. Further, programmers spend around 32% of their total time seeking information for executing their tasks [58].

Bailey conducted an experiment on a mix of students and professional. For studying the performance of a task, subjects were interrupted while they were executing their tasks. Bailey found that interruptions degrade the performance of tasks and the amount of degradation depends on the mental load that a programmer can take [49].

1.4.2 Studies Identifying the Differences Between Programmers

Grant and Sackman observed 12 programmers debug a given task [119]. Time taken by these 12 programmers was recorded. Analysis of the data showed that the time taken by the slowest programmer compared to that of the fastest programmer is in the ratio of 28:1. As the ratio of time taken by the fastest programmer to the slowest programmer can be heavily influenced by outliers, Prechelt used two metrics (Sf50 and Sf25) to study variations between programmers using a box plot [120]. Sf50 is the ratio of time taken at 75th quintile to 25th quintile, whereas Sf25 is the ratio of the time taken at 87.5th quintile to 12.5th quintile. Data collected from different experimental groups performing various types of tasks show that Sf25 is typically in the ratio of 2 to 3 and Sf50 is in the ratio of 1.5 to 2. Additionally, the variability (defined as the ratio of standard deviation to mean) of more than 0.5 is observed indicating that a programmer will take 50% more or less time than an average programmer [120].

Another study conducted by Gill suggests that productivity of programmer varies with experience [121].

Many other studies tried to identify differences between expert and novice programmers. Few observations from these studies are as follows:

- More abstract chunks of knowledge are seen in experts than novice programmers. Hence, expert programmers perform better while fetching meaningful information. Further, Expert and novice programmers comprehend program/software differently. Experts use mostly a systematic approach (top-down or bottom-up), whereas novices use opportunistic comprehension [122].
- Experts try to understand first the holistic/bigger picture of the problem and later decimate the problem into fine-grained smaller standard abstractions. Novices fail to integrate various parts of the problem description to appropriate knowledge structures [123].
- Novice finds it difficult to develop conceptual models regarding domain specific knowledge for understanding requirements [124].
- Novice programmers often find it difficult to fix compilation errors when compared with expert programmers [125].
- Expert programmers debug faster when compared with novice programmers [126].
- Experts automate simple components of the program [127].

- Experts make use of features (comments etc.) that help understand a program better. Hence, experts quickly navigate to the correct place for modifying the program [128].

Torii, et al. studied differences between experts and novice programmers using sophisticated data collection mechanism called "Ginger." Ginger had the capability of audio and video capturing of a programmer, video capturing of a computer monitor, eye tracking of programmer, skin resistance level sensors for measuring stress, etc.[129]. Their studies on experts and novices correlated well with the observations mentioned above.

1.4.3 Studies Trying to Modify the Work Practices of Programmers

Kersten and Murphy studied interaction logs of developers with IDE while developers switch from one task to another. They developed a tool to provide context switching information to programmers. They computed the productivity of programmers as a ratio of the amount of code-edits to the amount of browsing, navigating and searching. They have reported an improvement in productivity of programmers using their tool [130].

In another study, Hartmann first observed how expert programmers fix their compilation errors and then integrated the same in the development environment of novice programmers to help novice programmers fix their compilation errors. This study showed that novice programmers improved their error fixing by 47% [125].

Kim studied the effect of giving feedback to a programmer in a positive and negative manner on the productivity of programmers. Kim developed a tool called "TimeAware" which classifies applications used by programmers into five categories ranging from highly productive to least productive. Based on the time spent on applications, the tool gives feedback in a positive manner (80 mins out of 100 mins are productive) to few programmers and negative manner (20 mins out of 100 mins are non productive) to few programmers. Kim found that programmers who received feedback in a negative manner improved their productivity. However, the improvement in productivity did not sustain when the feedback from the tool is removed [131].

Marat Boshernitsan first conducted studies to identify how programmers reference code fragments while refactoring code regarding inputs, outputs and programming style [132]. Later, they have developed a tool to assist programmers in refactoring code. They found that programmers had completed their transformations quickly using the tool [133].

1.4.4 Personal Software Process

Personal Software Process (PSP) defined by Humphrey concentrated on improving individual programmer productivity. PSP is based on the principles mentioned below:

- Every programmer is different from others and hence their performances.
- To consistently improve once own performance, each programmer has to first collect and measure their data systematically and then create and implement plans for improving his/her performance based on the data.

To understand current performance, each programmer has to first record every job step that he/she perform along with the time spent on those job steps; and then baseline the captured data to lay action plans for any improvement. Although programmers can improve in many areas, PSP focuses mainly on reducing schedule deviations and minimizing defects.

PSP method is introduced in seven process versions labelled PSP0, PSP0.1, PSP1, PSP1.1, PSP2, PSP2.1, and PSP3 as illustrated in Figure 1.4. Each process version has a set of forms, scripts, logs, and standards to guide each programmer measure, record, baseline, and analyze his/her data.

PSP0 and PSP0.1 (Personal measurement)
PSP0 and PSP0.1 versions emphasize more on defining and recording existing process.

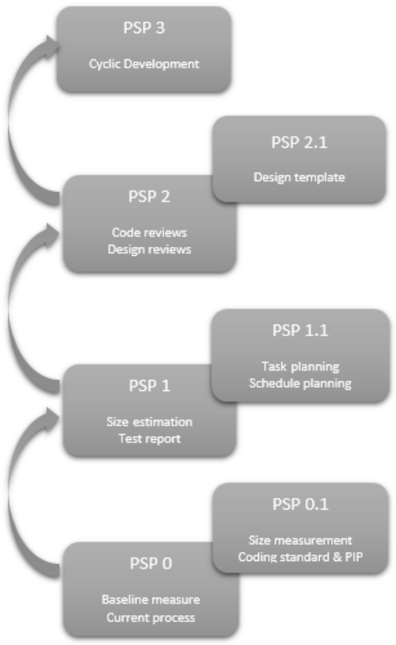


Figure 1.4: Personal Software Process

- *PSP0:* This version mainly concentrates on recording (baseline) the existing process. Programmers measure the time they spent in each phase of the process and logged all the defects captured during testing.
- *PSP0.1:* Based on the data recorded in PSP0, this version emphasizes to bring standard coding practices and measurement of the size of software.

PSP1 and PSP1.1 (Personal planning)

PSP1 and PSP1.1 lay importance on scheduling and task planning process (using the data of PSP0 and PSP0.1).

- *PSP1:* This version emphasizes on estimating the software size of a new task along with the defects that may arise (using the baseline data of size and defects in PSP 0 and PSP 0.1).
- *PSP1.1:* This version emphasizes on estimating the effort required to complete a new task using the size and defects estimated in PSP1. Scheduling and task planning are performed based on the estimated effort.

PSP2 and PSP2.1 (Personal quality)

Early defect detection helps to remove defects with less effort and minimizes schedule deviations. Therefore, PSP2 and PSP2.1 concentrate on the process of defect detection and removal.

- *PSP2:* Based on the defects (in PSP1 and PSP1.1), this version concentrates on bringing code and design review checklists to uncover most of the defects even before the start of the testing phase. This version help reducing schedule deviations by reducing the rework effort.
- *PSP2.1:* This version adds two more strategies “design specification” and “analysis techniques.” Programmers can improve their personal performance by measuring the time taken to complete their tasks and the number of defects injected/removed in each phase.

PSP3 (Cyclic development)

This is the final step in PSP and concentrates on scaling up the entire process to large projects through incremental software development.

1.4.4.1 Experience with PSP

Unlike CMMi, which is oriented toward project/organizational capabilities, PSP is more focused on improving individual programmer’s performance. PSP uses a systematic process (with checklist, forms, templates, etc.) to analyze one’s own capabilities (based on past data) to lay plans for improving quality and reducing schedule deviation. Effects of PSP has been well studied. Some of the findings are :

- The estimation accuracy of effort was improved. This was find in academic setting [60], as well as in industry [134].
- Programmers injected fewer defects and made the software more productive [61].

- High productivity students (who write more LOC/hour and inject fewer defects) reported that PSP made them less productive. On the other hand, students with low productivity, who inject more defects reported that PSP improved their performance [61].
- None of the students continued using PSP due to the overhead involved in collecting and analyzing data [46].
- Integrated tool support is required during practicing PSP; else, data quality problems can lead to wrong conclusions [42].
- Even with integrated tool support, accurate student behavior was not reflected [135].
- There was a significant improvement in the defect detection capabilities [136].
- Despite management guidelines, most of the programmers did not use full PSP process consistently [61].
- PSP cannot be applied as-is, but requires adaptations to match company practices [137].
- No industry programmer accepted to log the data at the required level of precision [138].

1.4.5 Pair programming

Programmers work in pairs to develop software or solve an issue/problem/bug. Continuous reviews and discussions between them trigger fast learning and help improve programmer productivity. However, there is no unified acceptance on programmer productivity improvement in pair programming. Some of the studies on impact of pair programming are:

- Gerardo and team stated that productivity of programmer improved when he/she moved to pair programming from solo programming [139].
- Pietinen and team stated that productivity of programmer reduced when he/she moved from solo programming to pair programming [140].

- Allen Parrish and team stated that the quality of the product improved in pair programming. However, the programmer productivity can reduce if the role-based protocol is not followed during pair programming [141].
- Novice-novice pair against solo novices is more productive than expert-expert pair against solo experts [142].
- Organizations has to perform cost benefit analysis for using pair programming in projects [143].
- An additional cost of 15% would be incurred to organizations for using pair programming. However, pair programming improves design quality and reduce defects [144].

Lack of good communication between the programmers could also hinder benefits of pair programming. Comparable experiences, capabilities, and high comfort levels between the paired programmers are a few factors that impact productivity in pair programming [145].

1.4.6 Other Studies

Few studies focused on how programmers interact with each other for developing new tools assisting them [146, 147, 148].

Many other studies concentrated on studying the effect of introducing/modifying a tool/process/method on productivity. For example:

- Whether or not test driven development is more productive than traditional software development [149, 150, 151].
- Whether or not agile software development yields an improvement in productivity [152].
- How adopting CASE tools in software development affect productivity [153].
- Effect of reuse on productivity [154, 155].

- The effectiveness of static analysis tools, use case templates, etc. on software quality and productivity [156, 157].
- Efficacy and efficiency of coverage criteria on software testing [158].
- Effect of code coupling on programmer productivity [159].
- Effect of two-person inspection on programmer productivity [160].

Mining software repositories is another area where the artifacts (developed by programmers) in the project repositories are analyzed for various objectives. A few of them includes but not limited to

- Giving useful insights to programmers and project managers for improving productivity, delivery, and quality [161, 162, 163, 164].
- Understanding and verifying taboos and assumptions in software development [165].
- Understanding overall software process [166].
- Identifying deviations/improvements in the overall software process [167, 168].
- Calculating project related metrics [169].
- Identifying the behavior of programmers [170].

1.5 Thesis Outline

As discussed in sections above, the developments in software processes have resulted in the systematic software development for achieving the project and organizational goals. Improvements in the overall software process increase software productivity by identifying and eliminating waste during the software development and optimizing the existing methods to reduce the software development effort. Over the years, research on software process has matured well with frameworks like CMMi, ISO, ASPICE, etc., which are used by many organizations to have highly matured overall software process.

Many studies have concentrated on how programmers comprehend software and identifying differences between novice and experts. Some studies looked into how programmers spend time at office on work and non-work related activities. Few studies tried to identify how programmers perform debugging, refactor a piece of code, task/activity switching's etc. None of the studies have carefully and systematically studied how programmers execute specific tasks, and how the method of execution of a task may impact the productivity of programmers. The work reported in this thesis focuses on doing a systematic study of how tasks are executed by programmers and how they affect productivity.

This thesis not only studies the programmer productivity at the task level, it also explores how the understanding obtained from these studies can be used to improve productivity of some programmers, particularly those who are not very productive. This is done not by self-learning through experience but by systematic learning from more productive programmers, and active effort of teaching the practices they use to others. This approach of study about how tasks are executed, how it impacts productivity, and how the learnings may be used to improve productivity of less productive programmers is a unique contribution of this thesis.

A software project consists of various tasks. A task is a piece of work and is always assigned to a single individual/programmer for its completion because further splitting of a task can be counterproductive. In a given matured overall software process (and tools/technology), most of the effort in a software project is consumed by programmers executing tasks assigned to them. Typically detailed schedule of a project shows the execution of a large number of tasks by programmers. Hence, for a given software process, the software productivity depends heavily on how efficiently the individual programmers execute various tasks. Therefore, there is a clear possibility of improving software productivity by improving individual programmer's efficiency in executing assigned tasks. This is the focus of the work reported in this thesis. The studies reported in this thesis were conducted in Robert Bosch Engineering & Business Solutions Ltd, a CMMi Level 5 software company, based in Bangalore, India.

Our study concentrated on improving the productivity of average-productivity programmers by identifying and transferring the best ways of executing a task to average-productivity programmers. The best way of executing a task is determined through the identification of differences between how high- and average-productivity programmers executed their assigned tasks. Our study is somehow in line with the concepts of Taylorism, value stream mapping, and standardization techniques mentioned earlier in this chapter.

Some of the differences identified between high- and average-productivity programmers showed that high-productivity programmers might not jump directly into the execution of task but spend a considerable amount of time understanding the assigned task initially. Further, high-productivity programmers seem to follow a systemic way of executing their tasks when compared to average-productivity programmers. These observations remind the studies mentioned earlier in this chapter between experts and novices to understand how experts and novices comprehend and execute their assigned tasks.

The rest of the thesis is organized as follows:

Chapter 2: ***Studying Task Processes***

In this chapter, we discuss what are task processes, and the framework to study task processes for improving programmer productivity. We first describe how we obtained the buy-in from the organization for conducting this research, as it is often a significant challenge. To study task processes of programmers, we use the tool "Snag-it" for video capturing the computer monitors of programmers. We then manually analyzed the task videos to identify task processes used by programmers. For the study, we grouped programmers into high- and average-productivity programmers - this we did using the feedback from the project managers as well as using the productivity data as measured in various tasks. This work was presented in Doctoral Symposium in ICSE 2014 [171].

Chapter 3: ***Impact of Task Processes on Programmer Productivity in Model-Based Testing***

In this chapter, we first discuss the testing in model-based software development [AST'14] [172] and then discuss the task processes used for unit testing in model-based development [ISEC'13] [173]. To understand the impact of task processes on the productivity of programmers, we studied the task processes used by programmers while executing testing tasks in a live model-based development project. Using our framework, we first identified two sets of programmers - one with high productivity and the other with average productivity. Then we studied testing tasks executed by these programmers to understand their task processes, and the differences between the task processes used by programmers in the two groups. We identified some differences in the task processes followed by these two types of programmers - both in terms of the steps performed as well as in terms of how the steps were performed.

Chapter 4: ***Modeling and Analyzing Task Processes***

In this chapter, we discuss how we model each task process as a network of steps and analyze them using Markov chains for a more formal analysis. For comparing two task processes, we compare the state transition matrices of the respective Markov Chains. The difference between two processes is quantified as the distance between their Markov Chains. We then studied the differences between high- and average-productivity programmers. We found that the task processes of high-productivity programmers were different from the task processes of average-productivity programmers. This work was reported in APSEC2015 [174].

Chapter 5: ***Impact of Transferring Task Processes of High-Productivity Programmers to Average-Productivity Programmers***

In this chapter, we discuss how we transferred task processes of high-productivity programmers to average-productivity programmers and the results that we obtained. We first identified the differences in the task processes between high- and average-productivity programmers and then transferred the task processes of high-productivity programmers to average -productivity programmers through training. We observed that the task processes and the productivity of average-productivity programmers improved. We also noticed that the similarity between the task processes of high and average productivity programmers improved. Our study indicates that it is possible to transfer the task process of high-productivity programmers to average-productivity programmers for improving their productivity.

Chapter 6: ***Other Studies for Improving Programmer Productivity***

Apart from studying the task processes of programmers, we also conducted few other studies on programmers for improving productivity. In this chapter, we discuss our other studies for improving programmer productivity.

The first study was to examine the effect of productivity of trainers on the productivity of new programmers. It is common in software development companies to employ a dedicated trainer to train and groom a new programmer. We asked high- and average-productivity trainers to train two new programmers each. We analyzed the productivity of new programmers after they had worked independently for at least six months after the completion of training. We found that the programmers trained by high-productivity programmers were more productive than the programmers trained by average-productivity programmers. Further, the task processes of new programmers were more similar to their respective trainers indicating that task processes can be transferred from trainers to trainees and has a significant impact on the productivity of programmers.

In the second study, we countered the "Parkinson's Law" behavior in programmers for improving productivity. When programmers are allocated more than required effort for executing a task, programmers may consume entire allocated effort and may not report "free time" in their weekly activity reports. But there are often issue resolution challenges due to which a programmer ends up wasting time. In our study, we allocated 33% less time to programmers than the estimated effort required for executing a task, and also facilitated a 15-minute time box for daily meetings to resolve issues that may impede the progress of a task. We found a significant improvement in productivity

[ISEC'13] [34].

In the third work, we studied the effect of model-based software development on the productivity of programmers during the maintenance phase of a project. In this study, we looked at many enhancement tasks in model-based development and compared them with enhancement tasks in traditional software development. We found that model-based software development gives at least 10% improvement in productivity [APSEC2014] [175].

Chapter 7: ***Summary and Discussion***

In this chapter we summarize the thesis, and discuss some possibilities of how the framework of using task processes for programmer productivity improvement can be employed, and the need for further studies. We have also discussed limitations and threats to validity of our study.

Chapter 2

Studying Task Processes

In a mature overall process with minimal wastage, the majority of effort in a software project is spent by programmers executing tasks assigned to them. Typically a detailed schedule of a project shows the execution of a large number of tasks by programmers. Hence, for a given software process, the software productivity is highly dependent on how efficiently the individual programmers execute various tasks assigned to them. Therefore, there is a definite possibility of improving software productivity by improving individual programmer's efficiency.

In this chapter, we will discuss the concept of tasks in a software project and also the concept of task processes - the processes used for executing these tasks. As a project is finally a network of tasks executed by people in the team, how efficiently these tasks are executed also have a due impact on the productivity of the project. Here, we will describe the framework we are using for studying task processes to understand their impact on programmer productivity. But, first we will describe how we motivated the need for this study, and got the buy-in from the organization. We have applied the framework mentioned in this chapter on programmers executing model-based unit-testing tasks. Our study design, approach, and results of model-based unit-testing are given in later chapters.

2.1 Getting the Buy-In

Empirical studies of actual practices in software engineering require observing programmers in an actual field environment. However, despite the existence of hundreds of software development organizations with thousands of programmers, due to the organizational and policy challenges that exist in business organizations, such studies are hard to conduct [176]. Consequently, empirical studies that requires direct observation of programmers are few [177].

Due to inherent uncertainties involved in exploring a research question, it is generally very hard for researchers to convince organizations for conducting empirical studies to address some research questions. This challenge gets compounded due to insufficient appreciation of value of research by managers and senior management, as well as due to business and deadline pressures. The situation gets further complicated in software services companies where projects are being done for customers, who are often directly involved in the projects, and whose approval for conducting a field study is also required.

Moreover, programmers and project teams also are not interested in participating in field studies, as it will necessarily involve some extra work, and the teams are under pressure for delivering quality software within schedule. Even when there is no schedule pressure, as their goals (often set at the beginning of the year) for performance evaluation generally do not include support for field studies etc., programmers and project teams are often not excited about participating in such studies.

So, if any empirical study is to be done on live projects in an organization, a strong buy-in is needed from multiple stakeholders. In addition, the study must comply with the policies of the organization - as most such organizations have strong policies and guidelines which severely limit what can be done.

For this work, to understand the policies and what is feasible, we first met Human Resource Department to understand the organizational policies and guidelines - mainly on capturing audio/video of programmers along with the computer monitors of programmers. We were told that the audio and/or video capturing of programmers is not allowed in the organization. Therefore, we dropped the idea of video taping of the programmers and agreed to work with video capture of their monitors. We then met the Legal department to get their opinion on the implications of capturing the computer monitors of programmers on the organization, and were informed that this is feasible with proper permissions and concurrence of project managers.

For doing such a study of task processes, it is extremely important to first establish its desirability and motivation. This requires not only articulating what the study is, but what benefit it may accrue to all the stakeholders, and take their inputs - which also helped them feel a part of the study. it is also essential to get a buy-in from Managers, as their support is critical since the experiment may involve extra work from teams. We had a discussion with the metrics team to understand the type of projects in the organization, and then obtained the email addresses of project managers from the IT infrastructure team.

To get inputs about the value of task processes study as well as get their buy-in, we conducted a Web-based survey of project managers to find out their views on studying task processes as a possible approach for improving programmer productivity. The survey was sent to 189 project managers, out of which 115 (over 60%) responded. Some of the key questions of the questionnaire and their responses are given in the Table 2.1.

Sl No	**Questionnaire**	**Yes**	**No**
1	Is your team working on embedded software?	69.30%	30.70%
2	Are your team members working on model based project?	50.50%	49.50%
3	Do associates in your team break a task into a sequence of sub-tasks (called task-process)?	91.90%	8.10%
4	Do your associates follow various task-processes in executing the tasks?	76.90%	23.10%
5	Task-processes followed by an associate in their tasks has an impact on his/her productivity	84.60%	15.40%
6	Studying the way an associate breaks a task into a sequence of sub-tasks can help identifying the areas of improvement	91.40%	8.60%

7	Studying the task-processes followed by associates in their tasks can help in identifying the best practices for executing tasks	84.80%	15.20%
8	Which tasks consume maximum effort in your project	–	–
9	Task-process that have positive influence on associates productivity	–	–
10	Task-processes that have negative influence on associates productivity	–	–

Table 2.1: Questionnaire for Survey of Project Managers

It was clear from this survey that project managers felt that the task processes used by a programmer (associate) impacts his/her productivity, and by studying task processes we can uncover best practices and can improve productivity of programmers.

In response to open ended questions (8, 9, and 10) we got many suggestions. While most were very general, as the project managers still did not know what task processes look like and what steps may be in them, the responses provided a useful background when studying task processes. The key suggestions are given below:

For Q8 (tasks that consume the maximum effort), most commonly stated tasks were: Analysis, Design, Testing, and Enhancements. Later, when we met with senior management of the relevant unit, it was agreed that the maximum number of people are involved in testing (and there were some projects which were exclusively doing unit testing), it is best to initially focus on this phase.

On the question regarding what task-processes have a positive influence, as expected, we got all types of replies, as it was not fully clear to them what activities are task-processes. It is still illustrative to list the nature of some of the common responses:

- Understand impact on other modules
- Identify and write unit test cases
- Do good analysis before coding

- Identify negative scenarios
- Self review
- Get design reviewed
- Clearly understand the tasks before starting
- Break down into sub tasks / develop a detailed work-breakdown-structure
- Micro-schedule your work
- Prepare text matrix before writing testcases

As we can see, all these reflect some type of practice that the project manager felt can help productivity while executing a task like unit testing of a module. This also supported our view that the steps or sub-tasks executed by a programmer when a task is assigned to him/her has an impact on the productivity. (For the last question, we did not get any interesting replies. However, one theme did show up frequently - doing multitasking was considered as having a negative impact.)

With this survey data, we met four vice presidents (business unit heads) and presented our research proposal along with the survey results to each of them separately. We also briefed them about our discussion with HR department and Legal department. We were told to focus on model-based projects as a majority of projects would be only model-based in future. Further, we were told to consider programmers with less than 3 years of experience as they are the major workforce in the organization. We also met the head of Quality and briefed him on the research, and identified the units where model-based development was happening. We met the respective senior managers and middle-level managers in the units for nominations of projects for our research.

We spent more than six months in this process. We feel that for such studies, researchers should be adequately prepared for this - the amount of effort that needs to be done to get a buy-in for conducting an empirical study in an organization is enormously high.

2.2 Task and Task Processes

Most software projects executed in commercial organizations have an overall plan that specifies the broad schedule with major milestones and is often obtained by estimating the effort and time for the project and the major components. This overall project plan is essential for planning and monitoring the project, but for execution and control it is of limited use for a Project Manager. For finer level control, a project is generally converted to a detailed schedule that is consistent with the overall schedule. The detailed schedule of a project, which typically evolves during project execution, typically has “tasks” that are generally assigned to one person [1, 178].

We will use the term task to refer to an activity assigned to one person (sometimes called a resource) in a software project and which has a clear deliverable or completion condition that is used by the project manager to assess the satisfactory completion of the task. In this thesis, we will focus primarily on tasks related to model-based testing - the task that consumes much of the effort in a model-based software project, and which was recommended by most project managers for producing improvement in the survey.

In most projects, a task may take a few hours to a few days to complete. Typically, a programmer assigned a task would execute it incrementally in small steps, each step performing some activity. We consider a step as performing some well-defined activity that will help in the completion of the task. How the execution of these steps is organized by a programmer is what we refer to as task process. In other words, a task process for a task is the sequence of steps used by a programmer for executing that task.

For example, for a task “develop functionality of a module X,” a programmer may use the following task process (specified as sequence of steps): build a skeleton, create basic functions, define methods and attributes, write code for a method, unit test the method, and repeat the last two steps as many times as required. As the overall software process generally does not specify or standardize task processes, so the task process can vary from one programmer to another. For instance, for the above task, a programmer may use the following task process: write a small piece of code, compile, fix compiling errors, repeat the steps till code for module X is done, test the entire code, fix uncovered bugs, and repeat testing and debugging till satisfied.

The above example illustrates that the task process for performing a same task may vary from programmer to programmer. Besides the structure of task processes in terms of steps employed and their order, the qualitative execution can also vary from programmer to programmer. For example, a programmer may write a few lines and then compile; another may write 10s or 100s of lines before performing the compile step; similarly, in the testing step, a programmer may execute one or two test cases, and another may execute 10s of test cases.

As the tasks in a project are determined in the context of the overall software process, the overall software process can be viewed as the organization of the execution of tasks. And the task process can be viewed as micro-processes, defining the process for execution of each of the task in the overall process. Or it can be viewed as a further detailing of the overall process by giving detailed steps in each of the tasks.

Task processes used by one programmer may vary from another as the overall software process does not standardize any task processes. On the other hand, we all know that some programmers are much more productive than others. So, the task processes used by programmers for performing the tasks assigned to them in a project is likely to have an impact on the programmer's productivity.

Therefore, there is a need to study the effect of task processes on programmer productivity. Particularly, we need to investigate questions like "What are the task processes used by programmers?" "What are the differences in task processes used by productive programmers from the task processes used by average programmers?" "Can we improve average programmer's productivity by transferring the task processes from the high-productivity programmers to average-productivity programmers?" Addressing such questions is the goal of this thesis.

2.2.1 Representing Task Processes

Task process is a sequence of execution of various small steps (some of which may be executed many times). Each step performs some well-defined activity toward completion of the assigned task and often resulting in some tangible output that the programmer later use. Associated with each step in a task process is some amount of qualitative and quantitative information.

For example, for a testing type of task, “writing new test cases” can be a step in a task process. Quantitative information associated with this step can be how much time was spent in doing this step. Qualitative information associated with this step includes how a programmer derived test cases, how expected output value of each test case is computed, etc. Qualitative and quantitative information on each step helps in effective analysis of the task processes of programmers. However, identifying and extracting information associated with each step is challenging.

A good amount of work has been done on modelling and analyzing business processes [179, 180]. Process models defined in the literature [181, 182, 183, 184, 185, 186, 187] focus on representing the overall software process.

At a basic level, we represent a task process as a table containing the sequence of steps executed by the programmer while executing the task. We can enhance each step in the table with qualitative (how was the step performed) and quantitative (duration of the execution) information associated with it. This table will have a limited number of steps, as a task eventually is completed. Some steps may be repeated many times in this sequence. Consequently, the number of distinct steps (or step types), represented by the name given to the step, will normally be much fewer than the number of steps performed in the task-process.

From this table capturing the sequence of steps executed, we can model the task process as a flowchart of steps - this will often be a compact graphical representation which can help in understanding the task process and for discussions. We sometimes used the flowchart representation in our initial studies.

While flowcharts can be a good way to represent and understand the flow of steps in a process, flow charts are not efficient to answer a query like "how similar are the two processes?" or what percentage of effort was used in executing a particular step. For this purpose, the tabular representation with attributes like frequency and duration spent are more suitable and were used. More on how this is done is discussed in the next Chapter.

However, the tabular representation also does not suffice if we want to ask a question how far apart are two task processes. For these type of questions, we later model a task process as a Markov chain where each step in the task process represents a node in the Markov chain. (More on this in chapter 4).

2.2.2 Comparing Task Processes

Task processes can be compared by considering steps, and qualitative / quantitative information of each step. In general, comparison of task processes is not easy due to the reasons like task processes are not defined precisely, and even when they are defined, as there are no commonly used standard process languages, it may not be defined as the same level of detail leading to vague comparisons.

Some quantitative comparison of task processes can be easily done by comparing their respective table of steps, or by comparing their flowchart representations. For example, to determine if there are different steps in the task process, the tabular or flow chart representation is sufficient. To study if there are differences in how many times a step has been executed or how much effort is spent in a step, then the tabular representation with its attributes about steps can be used (flowchart will not suffice, unless each node is augmented with attributes.) We will give examples of these representations later.

From the tabular representation, the process can also be represented as a Markov chain - where each step is a node and movement from one step to another is a transition. This representation is useful to quantify the difference between two task processes - the distance between the two Markov chains can be used as a measure of how similar/dissimilar are the two task processes. We will discuss this further in Chapter 4.

2.2.3 Framework to Study the Impact of Task Processes on Programmer Productivity

For studying the impact of task processes on the individual productivity of programmers, we took the following approach. We first identified some projects which had multiple programmers and which were of reasonable duration. We then identified two sets of programmers - high-productivity programmers and average-productivity programmers. Then we studied their task processes for the task under study (unit testing in model based development in our case), using which we identified the similarity between the task processes used by programmers in a group, and differences from the task processes used by the other group. The framework of our study is illustrated as a flow chart in Figure 2.1 - further details about each of the major steps in the framework is given below.

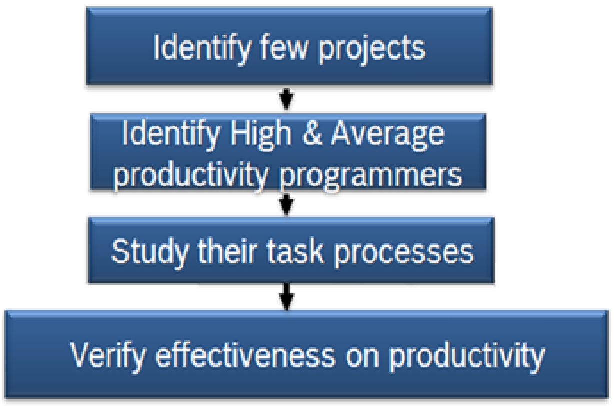

Identify few projects
Identify High & Average productivity programmers
Study their task processes
Verify effectiveness on productivity

Figure 2.1: Framework to Study the Impact of Task Processes on Programmer Productivity

2.2.3.1 Identifying Projects and Tasks for Studying

This step is important to analyze data between a group of productive programmers and a group of average programmers. We may not draw useful conclusions if programmers in this study belong to different types of projects. We say projects are of similar type based on domain, platform, language, type of tasks, overall processes, tools, development environment, etc.

To choose an area of work, project teams, and programmers, we conducted our study at Robert Bosch Engineering and Business Solutions Private Limited, a CMMi level 5 company, as discussed earlier in the Chapter. Based on that survey, the focus of this study was model-based software development projects, and within that unit testing of modules. Model-based software development is more promising to the organization and unit testing was already a major area of work in which a large number of programmers (particularly junior ones) were involved. Hence, the organization was keen to improve productivity in these projects.

The size of each project in the organization is around 12-14 members. Among these 12-14 team members, we could typically find 7-8 programmers (testers) have experience of fewer than three years. The remaining team members are experienced and seniors like project manager, associate project manager, architect, and specialists. Our framework requires grouping of programmers into two groups of high- and average productive which means that we need more number of programmers executing similar types of tasks. Moreover, based on the senior management inputs, we confined this study to programmers having 1-2 years of experience in the chosen area.

The projects were chosen which has many programmers working, so we can identify a group of high productivity and average productivity programmers. E.g. in the study described in the next Chapter, we selected a project which had many programmers and from them we selected six programmers with similar educational background, experience (1-2 years), and training - three of whom were classified as having high productivity and other three as having average productivity. As project managers are key to success of such experiments and we are relying on evaluation of project managers, these projects were those where project managers are with the team for a long time and know the performance of team members/programmers well.

For analyzing productivity in tasks, we need data on effort spent on the task and size for the task. For size, we decided to use whatever size measure is being used by the organization or the project (as it is a CMM level 5 company, there are organization level standards for size and all projects use size measures.) For measuring effort, the current measure used in the organization, which captures effort in hours, was inadequate for task level analysis. We will later describe how we obtained more refined effort measurements.

The project manager and programmers of the selected projects were first briefed about this study for a common understanding and their support and buy-in. For studying task processes, programmers were trained in aspects like self-recording, storing and analyzing data that was needed for the experiment. They were also engaged at all stages in review of the project and the information we collected on tasks or projects.

2.2.3.2 Identifying High- and Average-Productivity Programmers

Our framework requires identifying high- and average-productivity programmers in a project. High-productivity programmers are those who complete tasks (of appropriately same size and complexity) quickly and with good quality when compared to other programmers. That is, high-productivity programmers develop/test more code in a given amount of time than average programmers.

As task level past productivity data was not available, for grouping programmers into high-productivity and average-productivity programmers, we requested project managers to rate programmers on productivity as high or average.

This subjective evaluation by the project manager was later verified with the help of programmer's productivity data collected in the study. As it turned out, the initial assessment of project managers was accurate.

Both the programmer groups were briefed about the study, and their involvement, particularly in capturing data about the tasks. They were assured that this data will be used only for the experiments, all information about their tasks will be shared and verified with them, and details about their tasks or productivity will not be revealed to the management. Before giving this assurance to programmers, we had obtained this assurance from the senior management.

We did not inform programmers about our approach of classifying programmers into high and average productive groups. We told them that we are interested in studying the executions of tasks by programmers to uncover a best way of executing a task. However, we had shared our approach to management and project manager.

2.2.3.3 Studying Programmer's Task Processes

For studying task processes, we need to capture the effort data of programmers at a finer granularity than what is achieved by effort and activity logging systems, which are commonly used in many high maturity companies. Currently, for example, most effort data has a granularity of hours. Generally, a weekly activity report contains tasks that are assigned to the programmer, as well as the estimated effort for the assigned task (as estimated by the project manager). Once the task is executed, the actual effort is logged by the programmer. Besides assigned tasks, the report also contains Common Activities (like meetings, review, etc). All effort is recorded in hours - generally with a granularity of 0.5 hours. A sample of this is given in Table 2.2.

	Week 29.01.17 to 04.02.17						
Task Name	**29.01**	**30.01**	**31.01**	**1.02**	**2.02**	**3.02**	**4.02**
SlopeDetection_Coding (Scheduled)				8	8	8	
SlopeDetection_Coding (Actual)				6	9	10	
SlopeDetection_Rework_Coding (scheduled)						0.5	
SlopeDetection_Rework_Coding (Actual)						0.5	
Other task (Scheduled)							
Other task (Actual)							

Common activity -1 (Actual)							
Common activity -2							
Common activity -3							
Common activity -4							

Table 2.2: A Partially-Filled Regular Weekly Activity Report

We needed the effort data in minutes. For assisting programmers in recording the data, we first introduced the "Grind stone" tool [47] which they can use to record data. However, the collected data from programmers did not help due to various reasons: lack of uniformity across data for comparison, programmers still forgot to capture the data accurately or captured the data at a higher granularity than required, they often missed some key steps (or combined many of them) or the order of execution of various steps in the task, etc.

To obtain data at a much finer granularity for analyzing the task processes effectively, we decide to use the tool "snag-it" [188] to capture the screen shots of programmers monitors while executing the tasks. The tool captures one screen shot of the computer monitor per every 5 seconds of task progression. Then the captured screen is converted to a video, which can be played at a desired speed. With this video, one can see the screen of a programmer moving at a fast speed, and without actually watching the programmer. As the "video" is really the playing of the screen shots captured every 5 seconds, typically the video of a 4 hour session (which will have about 2880 frames) runs at normal speed for less than one hour. Processing this video to extract task processes initially is a slower process and takes a long time (a one hour video may take more than that to process). However, after processing videos of a few programmers, the processing becomes more efficient.

Before installing the tool, the programmers were assured that the captured videos would not be shared with anyone and were meant only for research purpose. Actual data collection for the experiment started only after a programmer became familiar with capturing of screen shots using the snag-it tool.

To identify steps in task processes of programmers and how they are organized from the video, we first identified the commonly used steps while executing a task based on discussions with the project teams. There is generally an existing common vocabulary that programmers use to discuss how they execute a task, which, with some refinements and standardization, is a good starting point for the set of steps that may exist in a task process. Actually, there also exists a broad general task process which has evolved over time based on past practices, project standards, and training, etc, which also helps define a common set of steps to be used and a general organization of these steps.

To ensure accuracy of observations, we also decided to analyze the video ourselves to identify the steps, their properties, and their execution sequence. While this required a huge amount of effort, we could not find any other way of getting the data at the level of accuracy that we needed for this study.

We first analyzed a task video of a programmer to identify what exactly he/she was doing. Generally, we grouped what appeared as a continuous activity perhaps with many minute activities as one (tentative step), and used visible transitions to a different type of continuous activity as transition to the next step. We noted these activities performed by a programmer in a table grouped together as one, along with its duration.

We then mapped these activities into steps (using the set of steps initially identified) - sometimes merging them where needed. If the activities grouped together had to be considered as separate steps - in that case, we had to go back and reanalyze the video. We discussed this analysis with the programmer to validate it and refined it based on his/her inputs. An example of some of the initial notes capturing the description of activities being done, along with the initial identification of steps, is shown in Table 2.3 (the complete notes for this had about 80 entries). As we can see, the initial notes had grouped the activities in a manner, but when identifying steps, sometimes multiple of these were merged into one step. And sometimes, they had to be divided among two steps.

Time (Hours : mm)	**Activities performed by programmer**	**Step name**
0:05	Open Bugzilla, check the versions of the files and documents, check for additional information if any	Analyze task request
0:10	Create folder structure, Open the tool ascet, open the version control system, check the required configurations of files, load the files, store the documents in the folder structure, Open the files in ascet	Load files
0:20	Check functional document, break the requirements in functional document into minute level in the excel sheet	Analyze functional document
0:05	Create test project model, delete the variables in the project	Setup test environment manually
0:05	Create calc and init tasks in the test project	
0:15	Create input tables in the test project, create expected output variable tables in the test project, create local variables for capturing values, give connections from timer to each table to model, put comparison model for comparing outputs and expected outputs	
0:10	Assign task numbers to all variables in the test project, update sheet of acceptance	
0:10	Set scheduler and assign priorities to tasks, import formulas, select signals for viewing	
0:10	Generate template for test spec tool, check functional document, import variables in test spec tool, add variable names in aspects, link aspect variables with variables in test cases, configure the path of storage	Create aspects
0:10	Write variable names in excel sheet, write time points, write test cases (2), rename variable names to suit to test spec tool	Write Test cases
0:10	Add variable names in aspects, link aspect variables with variables in test cases, configure the path of storage	Create aspects
0:03	Execute test cases	Run test cases
	Select signals (variable values over execution time) for viewing	Understand code behavior

	Analyze variable values for understanding the behavior of code	
0:05	Modify test cases, execute test cases	Modify test case inputs in excel; Run test cases
	Select and view signals, analyze variable values for understanding the behavior of code	Understand code behavior
0:05	Modify test case inputs	Modify test case inputs in excel
	Execute test cases	Run test cases
	View signals, analyze variable values for understanding behavior of code	Understand code behavior
0:10	Write variable names in excel sheet, write time points, write test cases (2), rename variable names to write 31 test case inputs, derive output for those test case inputs	Write test cases
0:03	Execute test cases	Run test cases
	View signals, , analyze variable values for understanding the behavior of code	Understand code behavior
0:04	Modify expected output variable values of two test cases	Modify oracles of test cases in excel
	Execute test cases	Run test cases
	Select and view signals, analyze variable values for understanding the behavior of code	Understand code behavior
0:15	Write time points , write another 30 test cases by looking into model parameters, calculate expected output values	Write test cases

Table 2.3: Initial notes of activities and steps identified from the task video

Identifying steps in a task process is an iterative process. As discussed above, we identified the sequence of steps initially using our judgement and understanding of the steps involved in the task process. We then verify this with the programmer, and make corrections to our understanding - for this we often view the video along with the programmer to gain a common understanding of the sequence of steps being performed. Frequently, we had to go over parts of the video and their analysis multiple times to gain this common understanding. This iterative process gives us the initial task process for that programmer for executing that task.

Sometimes, while analyzing task processes of other programmers, we will identify some steps that were earlier not used, or realize that what we considered earlier as one step should be considered as two or more distinct steps. For example, in one case we considered working with excel spreadsheets for test cases as "writing test cases" step. Later we realized that actually there are two other distinct steps involved in this - "writing test inputs" and "writing expected outputs". With this understanding we went back and refined the task processes suitably.

Overall, the process of identifying the steps and the task processes is iterative and tedious.

2.2.3.4 Analyzing the Impact on Productivity

To study the impact of task processes on programmer productivity, we first grouped task processes into two groups: task processes of all high-productivity programmers and task processes of all average-productivity programmers. From these tables, we can identify similarities in task processes of the programmers in two groups. As we noticed, the steps used by one group were mostly the same, with only a few differences. In fact, we noticed that the steps in the two groups were mostly the same, with only a few differences.

From these task tables, we obtained a consolidated table for the average and high productivity tasks capturing a summary of how the tasks of high- and average-productivity programmers use the steps in their task processes. From these tables, and the detailed tables of tasks, through observation we can identify some differences between the task processes of the two groups. However, as these differences were identified by us, we also verified the observations with the project managers and programmers. Initially we studied the productivity of the two groups, and also identified a few key differences based on these tables only. Results of this study are given in next Chapter.

To quantitatively study the impact of difference between task processes on productivity, we modeled the task process as a Markov chain and modeled the difference between them as the norm-1 or L1 distance. Then we studied the impact on productivity and other parameters with respect to the distance. Results of this are given in

Chapter 4.

The ultimate goal of studying task processes was to see if after studying the differences, can something be done to improve the productivity of programmers, in particular the average-productivity programmers. To study the usefulness of our task processes approach, we tried to transfer the task processes of high-productivity programmers to average-productivity programmers by training the average-productivity programmers to use high productive task processes. For the training, good understanding of similarities and differences in task processes of two groups was essential. We then collected new task videos from average-productivity programmers to study their new task processes and productivity. We again used Markov chains to compare old and new task processes of average-productivity programmers for investigating improvement in their task processes. Improvement in both task processes and productivity confirmed the usefulness of our approach for improving programmer productivity. More about this study is given in Chapter 5.

2.3 Summary

In this chapter, we defined the notion of task and task processes, and how a task process may be represented and compared. We motivated the need for studying task processes and also explained how we were able to get the buy-in from the organization, something that is extremely challenging but necessary for performing such studies. We also defined our approach of studying task processes for improving productivity of average-productivity programmers. In the next chapters, we will describe how this framework has been used for studying the impact of productivity in model-based testing.

Chapter 3

Impact of Task Processes in Model-Based Testing

To understand the impact of task processes on the productivity of programmers, we study how high- and average-productivity programmers execute their tasks while performing testing tasks in model-based software development. In this chapter we first discuss model-based testing and then discuss how we studied task processes of programmers in model-based testing for identifying differences in task processes between high- and average-productivity programmers.

3.1 Model-Based Software Development Process

A traditional software development process generally has these main phases: requirements, high level design, detail design, coding, unit testing, integration and system testing. (Activities for these main phases may be organized in different ways in different process models.)

A model-based development process differs somewhat from the traditional software development process. Requirements are generally done in a similar manner. However, in place of high level design, generally a meta-model is developed. During requirements and high level design, a functional document is often developed which contains information about inputs and outputs, constraints on them, and relationship among them. These capture the functionality. In place of detailed design, in model-based development, a detailed model is build for a unit/component. Detail modeling is done using formal language, and the detailed model is used to generate code which is then executed. Unit testing is done at the detailed model level (though for testing the generated code is executed.) Overall development process is shown in Figure 3.1.

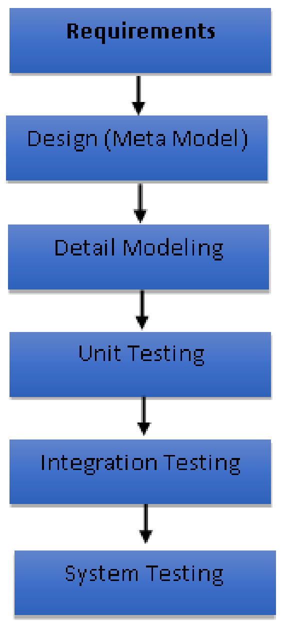


Figure 3.1: Overall Process for a Model-based Software Development

As we can see, this model differs from the regular software development process in the design and unit testing phases. Also, while coding and unit testing are often done by the programmer in a regular software development, in model-based development these are easy to separate and be executed by different persons. In Bosch, these two activities are normally done by different teams - one team develops the detailed model, while the other does the unit testing. Between modeling and unit testing, we can say that broadly the effort distribution is roughly 60:40. Unit testing is clearly an activity that consumes a considerable amount of resources. In Bosch, many teams focus exclusively on unit testing of detailed models, which are developed by other teams (which are sometimes located in a different location altogether.)

3.2 Unit Testing in Model-Based Software Development

Typically a model in model-based software development has a functional document that contains information about input/output variables (names, type, range, etc.) and parameters/constants (value, type, etc.) along with the functionality (functional requirements) of the model.

Model-based unit testing in Bosch includes writing test cases to test functional requirements of a model along with achieving 100% statement, decision and condition (basic and modified) coverage of code auto-generated from the model (for safety critical software in avionics and automotive domains). A sample model-under-test along with a portion of its functional document is given in Figures 3.2 and 3.3 that are used as a base to explain model-based unit testing.

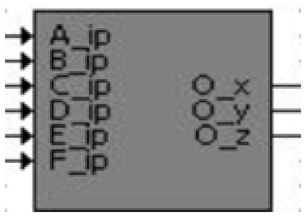
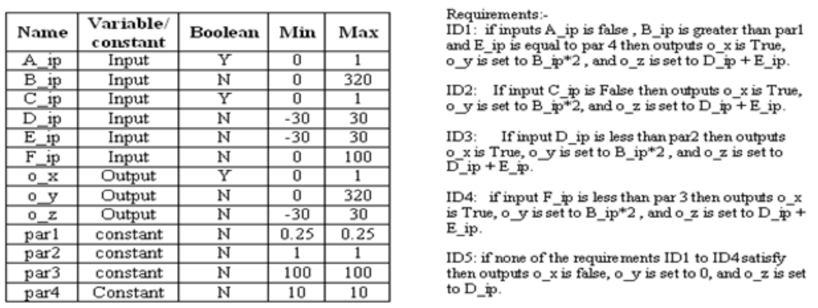


Figure 3.2: A Sample Model-Under-Test



Name	Variable/ constant	Boolean	Min	Max
A_ip	Input	Y	0	1
B_ip	Input	N	0	320
C_ip	Input	Y	0	1
D_ip	Input	N	-30	30
E_ip	Input	N	-30	30
F_ip	Input	N	0	100
o_x	Output	Y	0	1
o_y	Output	N	0	320
o_z	Output	N	-30	30
par1	constant	N	0.25	0.25
par2	constant	N	1	1
par3	constant	N	100	100
par4	Constant	N	10	10

Requirements:-
ID1: if inputs A_ip is false , B_ip is greater than par1 and E_ip is equal to par 4 then outputs o_x is True, o_y is set to B_ip*2 , and o_z is set to D_ip + E_ip.

ID2: If input C_ip is False then outputs o_x is True, o_y is set to B_ip*2, and o_z is set to D_ip + E_ip.

ID3: If input D_ip is less than par2 then outputs o_x is True, o_y is set to B_ip*2 , and o_z is set to D_ip + E_ip.

ID4: if input F_ip is less than par 3 then outputs o_x is True, o_y is set to B_ip*2 , and o_z is set to D_ip + E_ip.

ID5: if none of the requirements ID1 to ID4 satisfy then outputs o_x is false, o_y is set to 0, and o_z is set to D_ip.

Figure 3.3: A Sample Functional Document

The model-under-test given in figure 3.2 shows 6 inputs (namely A_ip, B_ip, C_ip, D_ip, E_ip, and F_ip) and three outputs (namely O_x, O_y, and O_z). Information about these input and output variables will be given in the functional document. For example, the functional document given in figure 3.3 shows the range of values that each input and output variables should take i.e., the input and output variables should not take values exceeding the range (Min - Max) given in the functional document. Functional document also contains the set of requirements that should have been ideally captured by developers in the model-under-test. The functional document in figure 3.3 shows five requirements (ID1 to ID5) which should be tested to check whether or not the requirements are captured fully in the model-under-test. Functional document also contains information about parameters and constants as shown in figure 3.3.

Similar to traditional software testing, we first create a test setup for executing test cases. The test setup is another model that includes model-under-test and an evaluation model as illustrated in Figure 3.4. The main purpose of the evaluation model is to automatically evaluate execution of test cases by comparing the expected and actual output variables of each test case.

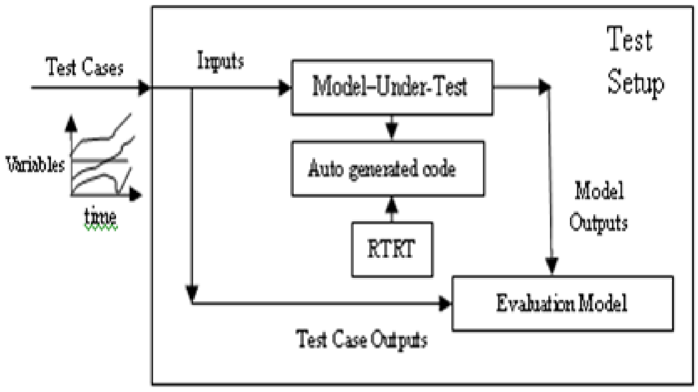


Figure 3.4: A Sample Test Setup

We present input and expected output values of each test case to a test setup, where the input values are applied to a model-under-test, and the expected output values are applied to an evaluation model. The evaluation model verifies the output values with the expected output values for each test case from the model-under-test.

In general, a test setup is run continuously over a certain time duration during which all test cases are applied continuously one after the other. The values of input and output variables of each test case are defined like signals varying with time. The evaluation model sets a flag when any test case fails (i.e., when expected output values are different from actual observed output values from the model).

It should be remembered that aim of this testing is to test the code auto-generated from model-under-test rather than the model-under-test itself as the auto-generated code only goes into the embedded controller (or electronic control unit). Therefore, when we run a test setup, auto-code generator integrated with the test environment generates code automatically from the model-under-test. The model-under-test is replaced with the auto-generated code such that the test cases are applied on the auto-generated code rather than the model-under-test.

To analyze test cases, the values of all variables/parameters/constants present in model-under-test and evaluation model are captured at every time point during the entire duration of test setup run. We can analyze the same (signals varying with time) visually. The flag updated by an evaluation model is used to find the exact point of time when a test case failed. The values of variables during that point of time are visually analyzed to interpret the results effectively.

Testing in model-based development is different from the testing in traditional software development. In model-based software development, highly optimized code is generated automatically from models. Such a code is often hard to understand and hence it is difficult to write test cases for it. Therefore, unlike traditional testing where test cases are derived based on a code to achieve coverage of the same code, model-based development requires test cases to be obtained based on the models and achieve coverage of code auto-generated from those models. Further, safety standards in automotive and avionics domains often demand effective testing methods to check functional requirements as well as achieve 100% coverage of code auto-generated from models. A few testing methods in model-based software development are described in the following subsections.

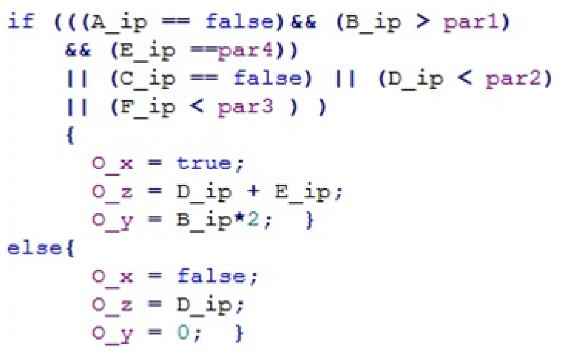
3.2.1 Modified Condition and Decision Coverage Method

Modified Condition and Decision Coverage (MCDC) method is used effectively in traditional software development and requires derivation of test cases based on the structure of code [189] to achieve 100% coverage on the code[190]. Test cases derived using the structure of code also check functional requirements [191]. MCDC method has proven to be very effective to uncover defects with less number of test cases [192, 193, 194]. Adrich tried MCDC method [195] in model-based software to test model coverage rather than auto-generated code.

Models in model-based software development capture a piece of code diagrammatically as expressions consisting of conditions and Boolean operators or as Boolean logic circuits. If required, Karnaugh maps are applied on the part of complex expressions to obtain a minimum Boolean expression. MCDC method derives just n + 1 test cases to effectively test a simple Boolean expression containing n conditions, and not $2\hat{n}$ test cases. In other words, we can derive only 4 test cases using MCDC to test a simple Boolean expression having three conditions like "a&b||c."

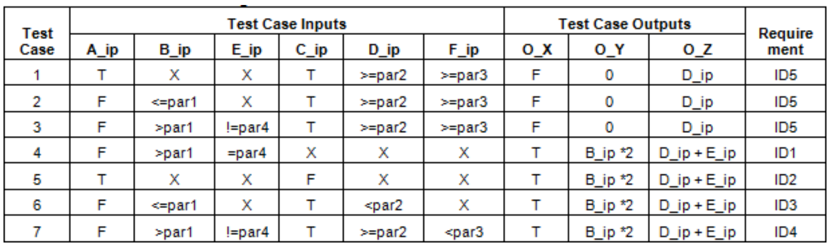
Test cases derived using MCDC method [196, 197] to test the Boolean expression shown in Figure 3.5 are shown in Figure 3.6. Don't-care conditions 'x' in Figure 3.6 suggest that there is no effect of input variable on output variable. For example: in test case 1, based on the evaluation order of conditions in the expression "(A_ip == false && B_ip>par1 && E_ip==Par4)", B_ip and E_ip are not checked when A_ip is already true. Similarly, E_ip is not checked if A_ip is false and B_ip<=par1.

It is evident from Figure 3.6 that test cases derived using MCDC method covered all functional requirements given in the function document (Figure 3.3). However, after assigning input variables with values in their min-max range (as mentioned in the functional document), coverage of auto-generated code is often not 100% as illustrated in Figure 3.7. This shows that the code auto-generated from a model do not have a direct mapping with the model. Further, coverage of statements, decisions and basic conditions are not 100% as auto-generated code adds extra statements to limit values



```
if (((A_ip == false)&& (B_ip > par1)
    && (E_ip ==par4))
    || (C_ip == false) || (D_ip < par2)
    || (F_ip < par3 ) )
    {
      O_x = true;
      O_z = D_ip + E_ip;
      O_y = B_ip*2;   }
else{
      O_x = false;
      O_z = D_ip;
      O_y = 0;   }
```

Figure 3.5: A Sample Auto-Generated Code



Test Case	Test Case Inputs						Test Case Outputs			Require ment
	A_ip	B_ip	E_ip	C_ip	D_ip	F_ip	O_X	O_Y	O_Z	
1	T	X	X	T	>=par2	>=par3	F	0	D_ip	ID5
2	F	<=par1	X	T	>=par2	>=par3	F	0	D_ip	ID5
3	F	>par1	!=par4	T	>=par2	>=par3	F	0	D_ip	ID5
4	F	>par1	=par4	X	X	X	T	B_ip *2	D_ip + E_ip	ID1
5	T	X	X	F	X	X	T	B_ip *2	D_ip + E_ip	ID2
6	F	<=par1	X	T	<par2	X	T	B_ip *2	D_ip + E_ip	ID3
7	F	>par1	!=par4	T	>=par2	<par3	T	B_ip *2	D_ip + E_ip	ID4

Figure 3.6: Test Cases Generated using MCDC Method

of variables within its defined range.

As MCDC test cases are usually not sufficient to achieve 100% coverage of auto-generated code, boundary conditions are incorporated on each input and output variables in the test suite mentioned in Figure 3.6. While incorporating boundary conditions we should not change the existing association of test cases with respective functional requirement(s).

To incorporate boundary value analysis for all non-Boolean input variables, there should be at least three test cases such that the value of the variable is set to greater than its max range in one test case, lesser than its min range in the second test case and equal to the parameter under comparison in the third test case. For Boolean input variables, there should be at least two test cases such that the value of the variable is set to "True" in one test case and "False" in the other test case. Further, don't-cares in the test suite should not be used (altered) while adding boundary conditions on input variables. Moreover, adding boundary conditions on output variables should preserve

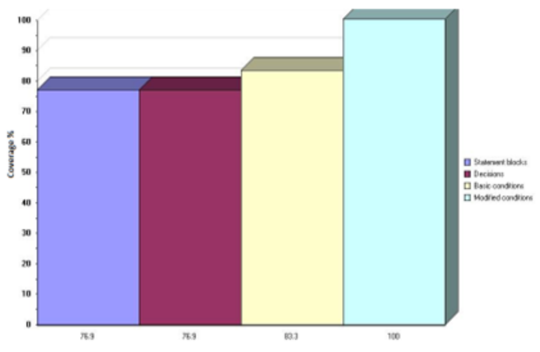


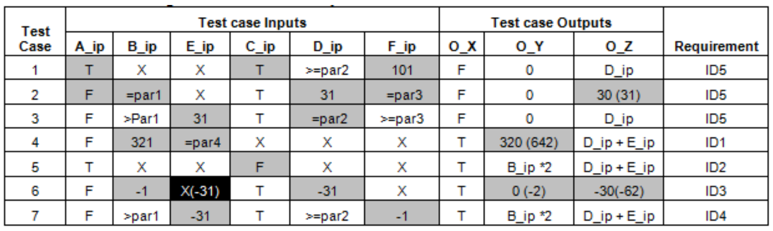
Figure 3.7: Coverage of Auto-generated Code using Test Cases Derived from MCDC Method

the boundary conditions on input variables. Further, don't-cares in the test suite are used to test the boundary conditions on output variable, that is, the values assigned to don't-cares of test cases can be utilized to test the boundary conditions.

In most of the cases, test cases derived using MCDC method are sufficient to add boundary conditions on input and output variables. However, in very few cases, some test cases have to be duplicated to bring boundary conditions on input variables.

After incorporating boundary conditions on the test suite given in Figure 3.6, the final test suite is provided in Figure 3.8. It is evident from Figure 3.8 that relationship between test cases and functional requirements are not altered. Each non-Boolean input variable has at least three test cases (in gray color in Figure 3.6) testing its boundary conditions, and don't-cares are not used to bring boundary conditions on input variables. Similarly, each output variable has at least two test cases testing its boundary conditions (in gray color). Further don't cares are utilized (in black color) to bring boundary conditions on output variables. For example, E_ip variable in test case 6 is a don't-care but it is used to check boundary value of output O_Z.

Performing unit testing using the test cases mentioned in Figure 3.8 gives 100% statement, decision and condition (basic and modified) coverage on auto-generated code as shown in Figure 3.9.



Test Case	Test case Inputs						Test case Outputs			Requirement
	A_ip	B_ip	E_ip	C_ip	D_ip	F_ip	O_X	O_Y	O_Z	
1	T	X	X	T	>=par2	101	F	0	D_ip	ID5
2	F	=par1	X	T	31	=par3	F	0	30 (31)	ID5
3	F	>Par1	31	T	=par2	>=par3	F	0	D_ip	ID5
4	F	321	=par4	X	X	X	T	320 (642)	D_ip + E_ip	ID1
5	T	X	X	F	X	X	T	B_ip *2	D_ip + E_ip	ID2
6	F	-1	X(-31)	T	-31	X	T	0 (-2)	-30(-62)	ID3
7	F	>par1	-31	T	>=par2	-1	T	B_ip *2	D_ip + E_ip	ID4

Figure 3.8: Boundary Test Cases Integrated with MCDC Test Suite

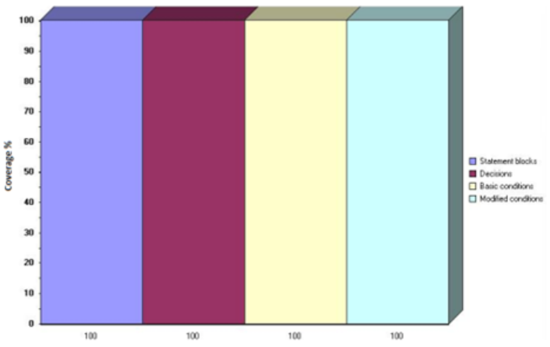


Figure 3.9: Coverage of Auto-generated Code using Test Cases derived from MCDC Method with Boundary Conditions

3.2.2 Classification Tree Method

Classification Tree Method proposed by Conrad systematically tests embedded model-based software [198]. This method requires first dividing the range of each input variable into a set of scenarios/aspects (or equivalent classes) and then carefully selecting one of the test scenarios of each input to frame a test case for testing a particular requirement. This method is highly effective to test functional requirements. However, the effect of this method on coverage of auto-generated code is not well understood.

Based on functional requirements, each variable input range is divided into a set of equivalent classes called aspects such that the behavior of code/model is same for any value within the range of an aspect. For example, based on the functional requirements given in Figure 3.3, input variable B_ip is dependent only on par1 and hence the range of B_ip can be divided into a minimum of two aspects: "<=par1 (<=0.25)" and ">par1(> 0.25)" such that behavior of the code remains same for any value within the range of each aspect. Similarly, A_ip and C_ip are Boolean variables and therefore can have only two aspects called False (0) and True (1). Aspects of all the input variables are shown in Figure 3.10.

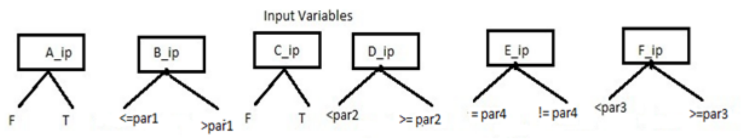
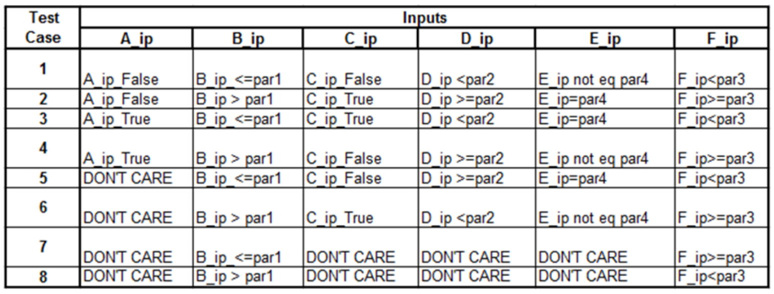


Figure 3.10: Equivalent Classes of Variables in Classification Tree Method

In general, combination rules (minimum criteria, maximum criteria and n-wise combination criteria) are used to combine aspects of input variables to generate test cases automatically. Test cases obtained automatically using pair-wise combination criteria are shown in Figure 3.11. For each test case, to calculate expected values of output variables, the aspects in the generated test cases are assigned a value representing the aspect. To get good coverage of auto-generated code, boundary values of input variables are assigned with respective aspects. Further, to satisfy all functional requirements, don't-cares have to be utilized. Code coverage obtained after utilizing don't-cares of test cases and assigning boundary values to aspects is given in Figure 3.12, which shows that coverage obtained is not 100% and hence require adding extra test cases to achieve 100% code coverage and test all functional requirements.



Test Case	Inputs					
	A_ip	B_ip	C_ip	D_ip	E_ip	F_ip
1	A_ip_False	B_ip_<=par1	C_ip_False	D_ip <par2	E_ip not eq par4	F_ip<par3
2	A_ip_False	B_ip > par1	C_ip_True	D_ip >=par2	E_ip=par4	F_ip>=par3
3	A_ip_True	B_ip_<=par1	C_ip_True	D_ip <par2	E_ip=par4	F_ip<par3
4	A_ip_True	B_ip > par1	C_ip_False	D_ip >=par2	E_ip not eq par4	F_ip>=par3
5	DONT CARE	B_ip_<=par1	C_ip_False	D_ip >=par2	E_ip=par4	F_ip<par3
6	DONT CARE	B_ip > par1	C_ip_True	D_ip <par2	E_ip not eq par4	F_ip>=par3
7	DONT CARE	B_ip_<=par1	DONT CARE	DONT CARE	DONT CARE	F_ip>=par3
8	DONT CARE	B_ip > par1	DONT CARE	DONT CARE	DONT CARE	F_ip<par3

Figure 3.11: Test Cases Derived using Classification Tree Method

Minimum combination criteria usually generate fewer test cases and require adding too many test cases manually for code coverage. On the other hand, maximum combination criteria generate too many test cases. Problems associated with too many test

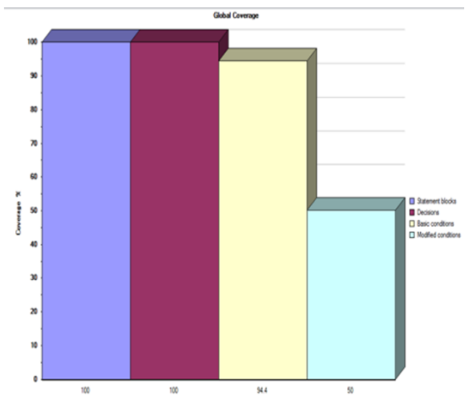


Figure 3.12: Auto-generated Code Coverage using Test Cases Derived from Classification Tree Method

cases include mapping test cases to functional requirements (traceability), redundant test cases, calculating expected output variable values of all test cases, etc. and make this criteria less viable.

3.2.3 Exploratory Testing Method

Exploratory testing is different from MCDC and Classification tree methods. Programmers derive new test cases by writing, executing and analyzing few test cases, and based on the results, they modify or overwrite the existing test cases by writing another set of test cases. In our research, programmers generally used exploratory testing.

3.3 Field Study - Analysis and Results

This field study was conducted at Robert Bosch Engineering & Business Solutions Private Ltd., a CMMi Level 5 software company that develops embedded software for the automotive domain and uses model-based software development extensively.

A model-based testing project was selected for this study. All the programmers in this project only test the tasks developed by a development team. Six programmers in the project were selected for this study. All these six programmers had one to two years of testing experience and had similar educational background and training for the project. The participating programmers were grouped into two groups of high- and average-productivity programmers, as discussed in the previous chapter. In this project, three out of six participating programmers were rated as high-productivity and the remaining three programmers as average-productivity.

Project managers assigned tasks to their programmers as per their project planning and scheduling process. Each programmer captured the video of at least two testing tasks and a total of 14 tasks were captured by the six programmers. Three high-productivity programmers captured seven testing tasks and the remaining three average-productivity programmers also captured seven testing tasks.

Apart from the task videos, we also collected the software size of tasks assigned to programmers. We did not introduce any new size measure specifically for this study but used “adjusted testable requirements (ATR)” as the size measure which was followed in the organization for a very long time. Project manager in this study used the same size measure and estimated the software size of tasks assigned to programmers as a part of their scheduling and task allocation process. ATR is a combination of the size measures “testable requirements” and “function points,” and the organization provides regular training on this size measure to project managers. Further, the organization does gauze R&R (repeatability and reproducibility) study repeatedly on the size measure.

It should be noted that the model-based testing tasks are often more challenging than testing in traditional development tasks because programmers have to generate test cases for the code auto-generated from models by analyzing the models directly. Further, as the tasks considered in this study belong to the application layer of the embedded automotive safety software, the variation between tasks is limited. The safety standards of the automotive domain have guidelines, which also restrict more variation in tasks. For example: in the software development of other domains, one task can have pointers, and another task may not have. One task may be calling for external files or repositories etc. but another task may not be doing that. The variation between tasks can be too much. However, when it comes to an application layer of safety embedded software, the difference between the tasks is less. They will neither use pointers, structures not call any additional file mechanisms, etc.

The tasks in this study are similar as they belong to the same application, same domain, same underlying platform (hardware), requires same tools and follows the same process. Further, the complexity of the tasks is also absorbed in the size of software estimated by the project manager.

3.3.1 Task Process of a Programmer

For understanding and analyzing task process of a programmer, the videos of each task were manually analyzed to first verify the steps involved in the task process and how they are organized, as discussed in the previous chapter.

To extract task processes of programmers, we further refined the tables extracted from the videos of tasks. We identified an initial set of types of steps that are likely to be in the task-process from the general definition of a task process that we had obtained earlier from the programmers. We also identified any new steps added by the programmer by looking at the videos and their analysis and confirming our understanding with the programmer. We finally obtained the task process as a table containing the sequence of steps executed for the task, and the duration of each step. An example of the task process of an average-productivity programmer (TP-A), is give in Table 3.1.

Duration (in minutes)	**Step**
0:05	Analyze task request
0:10	Load Files
0:20	Analyze Functional document
0:45	Setup Test Environment Manually
0:10	Create/modify aspects.
0:10	Write test cases
0:10	Create/modify aspects.
0:01	Run test cases
0:02	Understand code behavior
0:01	Modify test case inputs in excel
0:01	Run test cases
0:03	Understand code behavior
0:01	Modify test case inputs in excel
0:02	Run test cases
0:02	Understand code behavior
0:10	Write test cases
0:02	Run test cases
0:01	Understand code behavior

0:01	Modify oracles of test cases
0:02	Run test cases
0:01	Understand code behavior
0:15	Write test cases
0:02	Run test cases
0:01	Understand code behavior
0:01	Run test cases
0:01	Understand code behavior
0:01	Modify oracles of test cases
0:02	Run test cases
0:01	Understand code behavior
0:01	Modify oracles of test cases
0:02	Run test cases
0:01	Understand code behavior
0:10	Write test cases
0:02	Run test cases
0:01	Understand code behavior
0:15	Write test cases
0:02	Run test cases
0:01	Understand code behavior
0:05	Modify test case inputs in excel
0:02	Run test cases
0:01	Understand code behavior
0:01	Modify oracles of test cases
0:01	Run test cases
0:01	Understand code behavior
0:15	Write test cases
0:02	Run test cases
0:01	Understand code behavior
0:01	Modify test case inputs in excel
0:02	Run test cases
0:01	Understand code behavior
0:01	Modify oracles of test cases
0:02	Run test cases
0:01	Understand code behavior
0:01	Modify oracles of test cases
0:02	Run test cases
0:01	Understand code behavior
0:05	Write test cases
0:02	Run test cases
0:01	Understand code behavior
0:05	Write test cases
0:02	Run test cases
0:01	Understand code behavior
0:02	Modify test case inputs in excel
0:02	Run test cases
0:01	Understand code behavior
0:10	Derive oracles of test cases
0:01	Create/modify aspects
0:02	Run test cases
0:01	Understand code behavior
0:02	Create/modify aspects
0:02	Run test cases
0:01	Understand code behavior
0:10	Documentation
0:02	Analyze functional document
0:15	Documentation
0:03	Analyze test results
0:25	Documentation
0:08	Analyze code coverage
0:01	Modify test case inputs in excel
0:10	Write test cases
0:02	Run test cases
0:15	Update review checklists
0:01	Understand code behavior
0:03	Analyze test results

0:08	Analyze code coverage
0:10	Documentation
0:15	Update review checklists
0:15	Archive
0:05	Close task

Table 3.1: Task process of a task (TP-A) executed by an average-productivity programmer

As we can see, in the task process TP-A, the programmer executed 89 steps using 17 distinct steps, and took a total of 428 minutes (over 7 hours). We can also see that "run test cases" (and "understand code behavior") are the most frequently executed steps.

As discussed in previous chapter, obtaining this table representing the task process from the video is an iterative process. Generally, in the first instance, from the video a table will be prepared in which different phrases and terms will be used to express the different steps, and some information may also be captured while noting down the step. E.g. in some place one may note down "write test case" and in another it may be written "developing a test case". Another example: one entry of the table may be "wrote a test case" and another may be "wrote 10 new test cases". An example of this table was given in the previous chapter. From this initial table prepared from the video, the table 3.1 is extracted - in this table a standard set of names of steps is used. And information about the step is separated out.

Let us now see the task process of a high-productivity programmer. The task process for one task (TP-H) is given in Table 3.2. As we can see, for this task, the programmer executed 41 steps (of 17 distinct steps) and spent a total of 351 minutes (approximately 6 hours).

Duration	Steps
0:10	Analyze task request
0:20	Analyze Functional document
0:15	Load Files
0:40	Setup Test Environment
0:01	Analyze Functional Document
0:06	Create/modify aspects.
0:05	Analyze Functional Document
0:15	Copy Information to Excel
0:02	Automation (excel formulas) for output variable values
0:05	Copy Information to Excel

0:05	Analyze Functional Document
0:05	Write test case inputs
0:02	Automation (excel formulas) for output variable values
0:30	Write test case inputs
0:05	Run test cases
0:04	Understand code behavior
0:02	Write test case inputs
0:02	Run test cases
0:05	Understand code behavior
0:02	Automation (excel formulas) for output variable values
0:01	Analyze Functional Document
0:15	Write test case inputs
0:02	Run test cases
0:01	Understand code behavior
0:06	Write test case inputs
0:02	Run test cases
0:02	Understand code behavior
0:02	Analyze test results
0:08	Analyze code coverage
0:02	Modify test case inputs
0:10	Write test case inputs
0:01	Run test cases
0:01	Analyze test results
0:08	Analyze code coverage
0:40	Documentation
0:40	Update Review work/checklists
0:01	Run test cases
0:08	Analyze code coverage
0:05	Update Review work/checklists
0:10	Archive
0:05	Close task

Table 3.2: Task process (TP-H) of a task executed by high-productivity programmer

As we can see, though the starting point and end point of TP-A and TP-H are the same, the way these task processes progressed was different. Further, from our interaction with programmers it became clear that the differences in their tasks execution may also lie not just in the nature of steps executed but also in the manner how different steps were executed. Hence, besides identifying the steps and their organization, we also noted some qualitative information about how each step is executed. For this, as mentioned before for each step in every task process we also captured some attributes. We extracted the same from the captured videos. For example, for the step "write new test cases," the attributes include, the number of test cases after finishing each iteration, the time taken in each iteration, how test cases have been derived, the total number of test cases and the total time taken, etc. Similarly, for the step "setup test project," the attributes include "time spent in an iteration, what has been modified/added, overall time spent in this step, etc."

From these tables, we can also get summary information - e.g. how many times a step is executed, and how much time is spent on that step in the task process. This summary for the two task processes TP-A and TP-H is given in Table 3.3. As we can see, there are some steps that were not performed by the average productivity programmer (e.g. Copy information to excel and Automation). We can also see that time spent on some steps (e.g. run test cases or understand code behavior) is more than 2 times in TP-A, while it is significantly lesser in some other steps (e.g. Analyze task request and Analyze functional document).

Steps	**TP-H**		**TP-A**	
	Frequency	**Duration**	**Frequency**	**Duration**
Analyze task request	1	10	1	5
Analyze Functional document	5	32	2	22
Load Files	1	15	1	10
Setup Test Environment	1	40	1	45
Create/modify aspects.	1	6	4	23
Copy Information to Excel	2	20	0	0
Automation (excel formulas) for output variable values	3	6	0	0
Write test cases	0	0	9	95
Write test case inputs	6	68	0	0
Modify oracles of test cases	0	0	6	6
Derive oracles of test cases	0	0	1	10
Run test cases	6	13	23	42
Understand code behavior	4	12	23	27
Analyze test results	2	3	2	6
Analyze code coverage	3	24	2	16
Modify test case inputs	1	2	6	11
Documentation	1	40	4	60
Update Review work/checklists	2	45	2	30
Archive	1	10	1	15
Close task	1	5	1	5

Table 3.3: Summary of frequency and duration for TP-A and TP-H

3.3.2 Task Process of Average- and High-Productivity Programmers

To discuss the difference between the task processes of average- and high-productivity programmers, in this study, we studied the different task processes to identify some differences, and then discussed these with the programmers and managers to validate our findings. For facilitating this, we noted all the steps that are there in the task processes, and then counted the number of task processes that use it, as well as the averages of frequency and time spent in the steps by the task processes of the average- and high-productivity programmer groups. Table 3.4 gives the average frequency and time spent on some of the key common steps. For the table 3.4, following are used:

- **#TP:** Number of task processes that used the step
- **Avg Freq:** Average number of times a step is executed in a task process (determined as the number of times used in all the task processes / total number of task processes)
- **Tot Time:** Average total time duration spent on the step by a task process (in minutes)

Steps	**Average-Productivity Programmers**			**High-Productivity Programmers**		
	Total Task Processes = 7			**Total Task Processes = 7**		
	#TP	Avg Freq	Tot Time	#TP	Avg Freq	Tot Time
Analyze task request	7	1	6.4	7	1	7.85
Analyze Functional document	7	4.85	29.28	7	3.42	30.71
Load Files	7	1.28	15.71	7	1	12.85
Setup Test Environment	7	3.71	47.14	7	1.28	37.85
Create/modify aspects.	7	3.71	27.57	7	1.14	13.42
Copy Information to Excel	0	0	0	7	1.42	21.85
Automation (excel formulas) for output variable values	1	0.28	1.5	6	2.14	14.57
Write test cases	6	6.28	67.15	1	0.85	8.14
Write test case inputs	1	1.42	11.42	6	4.85	55.14
Modify oracles of test cases	5	3.14	16.57	0	0	0
Derive oracles of test cases	2	0.428	2.85	1	0.14	0.71
Modify test cases	4	2.28	11.71	0	0	0
Modify test case inputs	3	1.71	5.15	7	1.71	9.45
Run test cases	7	15.85	35.4	7	7.14	15.14

Understand code behavior	7	15.14	45.71	7	6.42	24.57
Analyze test results	7	2.14	8.57	7	2.14	7.85
Analyze code coverage	7	2.42	22.14	7	2.42	22.4
Documentation	7	2.42	65	7	2.42	44
Update Review work/checklists	7	1.57	32.87	7	1.57	33.57
Archive	7	1	17.14	7	1	14.28
Close task	7	1	5.28	7	1	5.42

Table 3.4: Averages for the task processes of average-productivity and high-productivity programmers

We can see from the table 3.4 that there are steps that are used by all (or almost all) high-productivity programmer, but are not used at all (or used in very few tasks) by average-productivity programmer (e.g. Copy Information to Excel, Automation, Write test case inputs, Modify test case inputs). Similarly, there are steps which are widely used by average-productivity programmers but are not used much by high-productivity programmers (e.g. Write test cases, Modify oracles of test cases, Modify test cases).

We also can see that even without including the steps whose usage differs too much, there are some steps in which high-productivity programmers spend much less time as compared to average-productivity programmers (e.g. Create/modify aspects, Run test cases, Understand code behavior).

These observations show that not only do the task processes for two groups of programmers differ in steps they execute, they also differ qualitatively in how the step is executed or how much time is spent on them.(The task process table for the different programmers can be made available to a researcher upon request under assurances of confidentiality.)

3.3.3 Productivity Difference between High- and Average-Productivity Programmers

We determined the productivity of programmers of the two groups for the tasks we studied. Results are shown as a box plot in Figure 3.13. As the data was normally distributed (as per Anderson-Darlington normality test), two sample t-tests showed that the project manager had rated their team members accurately on a productivity scale [high, average] with a statistical significance value of 0.001. This analysis confirmed that the project manager's evaluation regarding productivity was consistent with the actual productivity data. It also showed that the average productivity of the two groups differed by a factor of two. (This data also sheds some light on the issue of how reliable are subjective evaluation by project managers about technical competence of engineers working in the project).

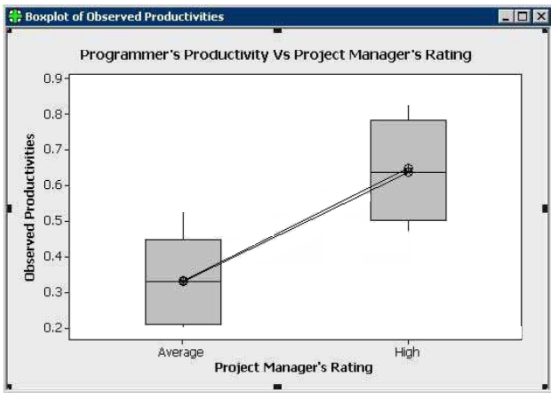


Figure 3.13: Productivity of Programmers in Two Groups

3.3.4 Differences in task Processes between High- and Average-productivity Programmers

We studied the task processes of high-productivity programmers as well as the summary tables for those. We identified the steps (and attributes) that are commonly used by high-productivity programmers and least used by average-productivity programmers, and vice versa. We computed the mean difference of the attributes between high- and average-productivity programmers. Apart from identifying the differences in the steps and their attributes, we also identified the steps that are often revisited by average-productivity programmers and less revisited by high-productivity programmers and vice versa. We also verified these observations with project managers and programmers.

In terms of the steps used in the task processes, there was one main difference we identified between the average- and high-productivity programmers' processes. And that was that high-productivity programmers have extra steps before writing test cases. As we have noticed before, almost all high-productivity programmers had these two steps (which average-productivity programmers did not have) - Copy information to Excel, and Automation (excel formulas) for output variable values (these are clearly related, though distinct steps). In these steps based on their analysis of the design document and requirements, they write an Excel formula to determine the expected output for the test case inputs. So, in step write new test cases they have to only decide the test case inputs. In essence, they perform an extra step to set up some automation for testing. This extra step validates the expected output and so they end up spending much less time in the later steps like Modify oracles of test cases, Modify test cases, Write test cases, Run test cases, Understand code behavior, etc. If a test case fails, all they need to do is check (and update, if needed) the Excel formula. On the other hand, an average-productivity programmer does not perform these steps and determines the expected output manually when writing the test cases, thereby spending more effort in many later steps.

For using automation, they also had a step of consolidating information for testing by copying them in excel. All the high-productivity programmers moved all the required information (like variable names, min-max values of variables, constant values, etc.) from various documents to an Excel sheet, and avoided referring multiple documents while writing test cases. This approach was not observed in average-productivity programmers. In other words, high-productivity programmers had an additional step in their task process that was not followed by average-productivity programmers.

Differences were also observed in how the steps were executed. Here we list a few such differences that we found. These differences are computed by comparing the attributes of steps used by high- and average-productivity programmers. A list of few attributes captured in a task process of a high- and average-productivity programmers are given in Table 3.5.

	High-productivity programmer task process	**Average-productivity programmer task process**
Total number of test cases derived by programmer	148	200
Total number of test executions	5	14
Total number of test cases written before each test execution	37,131,148,148,148	2,19,59, 129,129,129, 142, 200,200,200,200,200,200,200
Derivation of test case outputs	Excel formula (automation)	Manual
Test project setup	Use macros (Automation)	Manual
Derivation of test case inputs	Systematic (like MCDC)	Exploratory
Loading files into test setup	Loads only required classes	Loads entire module configuration
The number of times the step "test project setup" was visited after first test execution	1	1
The number of times the step "analyze functional document" was visited after first test execution	1	3
The number of times the step "understand code behavior" was visited after first test execution	6	16
Time spent on analyzing the steps "analyzing functional document" and "understanding code behavior" after first test execution	45mins	124 mins

Table 3.5: Attributes in the task process of a high- and average-productivity programmer

By using excel formula, programmers derived the expected output values for each test case automatically, and we believed that this had a significant impact on their productivity. That is, programmers who used Excel formulas (for fitting a relationship between output variables and inputs of the module) for generating the expected values of output variables in test cases were more productive than programmers who did not use Excel formulas, and use of the formulas was a key reason for their improved productivity.

To verify our observation, the productivity of programmers was grouped according to the way they derived expected values and the same is shown in Figure 3.14. As the data was normally distributed (as per Anderson-Darlington normality test), two sample t-tests showed that usage of Excel formulas during generating test cases had a significant impact on programmer productivity with a P value of 0.073.

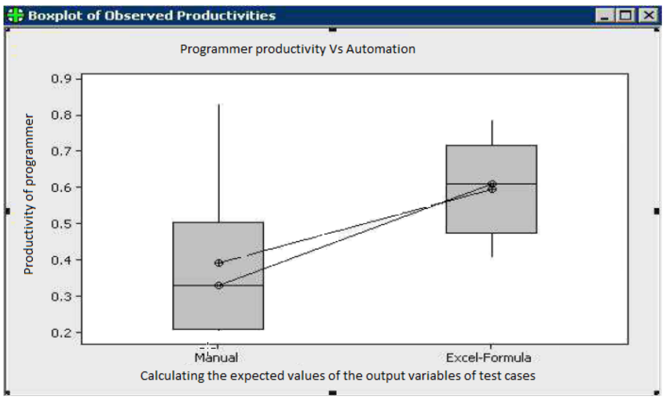


Figure 3.14: Productivity of Programmers Vs Automation Used by Programmers

In other words, high-productivity programmers used automation to determine expected outputs, while average-productivity programmers did not do so and this factor was the main contributor to productivity difference. This translated to having another activity in the high-productivity programmer's task process, which also made their task process "front heavy" as compared to average-productivity programmers. But this front-loading pays off.

- Average-productivity programmers distributed the analysis part throughout the execution of the task, whereas high-productivity programmers concentrated more on analysis part upfront even before writing test cases (resulting in the extra step of writing a formula).The table 3.5 suggest that a high-productivity programmer had visited the steps Analyze functional document and Understand code behavior only six times after the first test execution whereas an average-productivity programmer had visited these steps sixteen times after the first test execution.

- During the project setup step, all the average-productivity programmers loaded most of the library modules to test them, whereas the high-productivity programmers loaded only the required library modules into the test project setup.

- Average-productivity programmers wrote more test cases compared to high-productivity programmers to achieve the same code coverage set for the chosen project. (Number of test cases written was obtained from the qualitative information captured.) To verify our observation, we normalized the total number of test cases written by programmers in each of the tasks with respective software size of the task. This data for high-productivity and average-productivity programmers is shown in Figure 3.15. As the data was not normally distributed (as per Anderson-Darlington normality test), Kruskal-Wallis non-parametric test was conducted after removing the outlier. The results showed that average-productivity programmers wrote more test cases compared to high-productivity programmers with a statistical significance value of 0.048. The quality of the tasks developed by programmers was checked and it was found that all the tasks met and satisfied project quality standards.

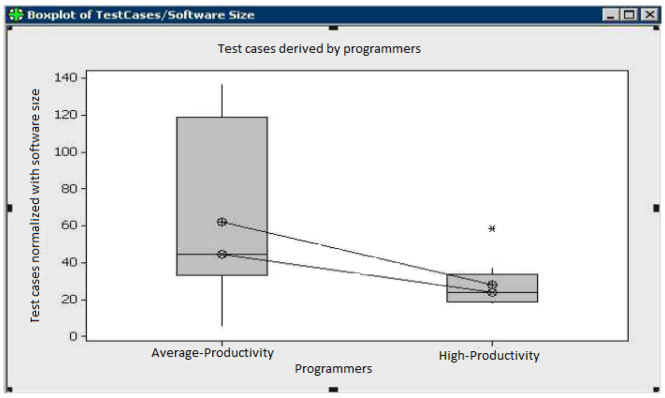


Figure 3.15: Number of Test cases Derived by Programmers

- Average-productivity programmers also used more iterations of writing a few new test cases followed by compiling and running test project setup, analyzing test results, modifying/updating test cases before writing another few new test cases over the already written test cases compared to high-productivity programmers. The number of executions of the test cases during the execution of assigned tasks was counted and the average is given in the table above. Subsequently, these executions were normalized with software size of the task and the box plot is shown in Figure 3.16. As the data was normally distributed (as per Anderson-Darlington normality test), two sample t-tests showed that average-productivity programmers performed more test executions compared to high-productivity programmers with a statistical significance value of 0.019.

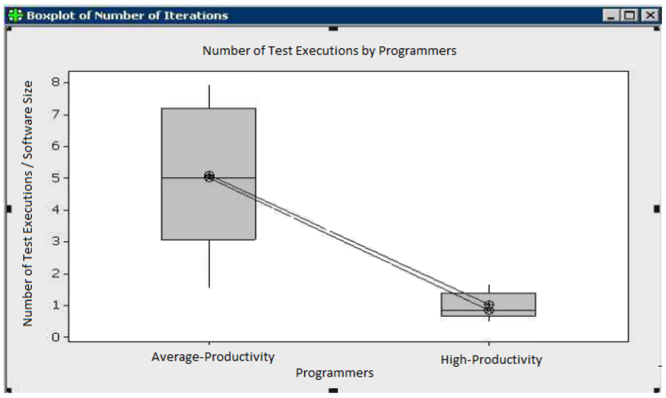


Figure 3.16: Number of Test Executions (Normalized with Software Size) by Programmers

3.4 Summary

We studied the task processes used by programmers while executing testing tasks in a live model-based development project in a CMMi Level 5 software company. To study the influence of task processes on productivity, we first identified two sets of programmers - one with high productivity and the other with average productivity. This identification was done subjectively by the project manager and later validated through productivity data. Then we studied testing tasks executed by these programmers to understand their task processes, and the differences with the task processes used by programmers in the two groups. We observed differences in the task processes followed by these two types of programmers - both in terms of the steps performed as well as in terms of how the steps were performed. This study indicates that task processes used by programmers during execution of tasks vary and the task processes employed by high-productivity programmers are different from the task processes used by average-productivity programmers, which suggest that task processes have an impact on programmer productivity.

Chapter 4

Modeling & Analyzing Task Processes

In the previous chapter we discussed task process of model-based testing, and some key differences between the task processes used by high-productivity and average-productivity programmers. For identifying the differences, we represented the process as a table and identified the differences by studying the task process manually and subjectively.

In this chapter, we will develop a more precise model and approach to compare task processes. For this formal analysis, we model the task processes of each programmer as a Markov chain. Each step in a programmer's task process is a node in the Markov chain. Further, we assume that programmer's next step generally depends on the execution outcome of the present step - this usually holds and is a criterion for applying Markov chains. For comparing two task processes, we compare the state transition matrices of the respective Markov Chains. The difference between two processes is quantified as the distance between their Markov Chains. We then applied this approach to study the differences between high- and average-productivity programmers.

4.1 Modeling Task Process

4.1.1 Task Processes and Set of Steps

For comparing Markov chains, we need to have the same set of states for all Markov Chains. This we did by first determining a full set of steps involved in the task processes. The table essentially enumerates all possible steps. In other words, it defines the vocabulary/names of steps, and is given in Table 4.1 for this study. This vocabulary is used to define each of the task processes. All these are reviewed with the programmers and managers.

Sl. No	**Step**	**Description**
1	Analyze task request	Programmers analyze their assigned task in the tool "Bugzilla" to understand prerequisites (like name and version of model-under-test, folder paths of respective functional documents etc.) for starting the task.
2	Setup test environment manually	Programmers create test environment manually
3	Setup test environment using macros	Programmers use various macros to create test environment.
4	Load Files	Programmers load required files from version control system into test environment
5	Analyze functional documents	Programmers go through functional documents to understand the requirements of model-under-test.
6	Compare functional documents with model-under-test	Programmers compare requirements with the functionality of model-under-test.
7	Copy information to excel	Programmers copy information (names of input variables, output variables, parameters etc.) from functional documents, model-under-test etc., to an excel sheet for writing test cases.
8	Write test cases inputs in excel	Programmers write test cases inputs. However, this step does not include calculating expected output values (oracles) of test cases

9	Derive oracles in excel manually	Programmers derive expected output values (oracles) manually for each test case in excel sheet.
10	Derive oracles in excel using excel formulas	Derive oracles in excel using excel formulas and Programmers just fit an excel formula in excel sheet for deriving output values of each test case
11	Create/Modify aspects in testing tool	Programmers have to write test cases in a separate testing tool that integrates with model-under-test for running test cases. This testing tool also generates test reports automatically. But this testing tool requires test cases to be written in a specific format that involves creating aspects of inputs and outputs of test cases.
12	Run test cases and generate reports	Programmers execute test cases and generates test reports using testing tool.
13	Analyze code coverage report	Programmers analyze the code coverage reports of test cases executed.
14	Analyze test results	Programmers analyze test reports to understand passed and failed test cases.
15	Analyze coverage of test cases	Programmers analyze the code coverage/execution paths of one or few particular test cases.
16	Understand code behavior	Programmers analyze variables in model-under-test to analyze particular test case.
17	Modify oracles of test cases in testing tool	Programmers modify the oracles of test cases in testing tool.
18	Modify test cases in testing tool	Programmers modify both the inputs and the outputs (oracles) of test cases in testing tool.
19	Modify excel formulas in excel	Programmers modify excel formulas to change the output values (oracles) of test cases.
20	Modify test case inputs in excel	Programmers modify the inputs of test cases in excel sheet.
21	Write test cases	Programmers write new test cases (both inputs and oracles) on top of existing test suite.
22	Modify test cases in excel	Programmers modify both the inputs and outputs (oracles) of test cases in excel.

23	Copy test cases	Programmers copy test cases from excel sheet to testing tool.
24	Documentation	Programmers add/modify the documentation of test cases.
25	Update review checklists	Programmers review all the deliverables of task using review checklists.
26	Archive	Programmers archive all the deliverables of task in version control system.
27	Close task	Programmers close task in the tool Bugzilla.

Table 4.1: Steps in Task Processes of Programmers Executing Model-based Testing Tasks

With this table of all the steps, the task process can be defined using this vocabulary. The task process for an average-productivity programmer is shown below in Table 4.2. In this table, the step no is the reference to the number of step in the Table 4.1. As we can see, in the execution of this task, the average-productivity programmer has used most of the steps, but not all (e.g. steps 3, 7, 10, 13, 16, 19, 20 are not used).

Sl. No	**Step Name**	**Step. No**
1	Analyze task request	1
2	Setup test environment	2
3	Load files	4
4	Analyze functional documents	5
5	Create/modify aspects	11
6	Compare functional documents with model	6
7	Write test cases inputs in excel	8
8	Analyze functional documents	5
9	Write test cases inputs in excel	8
10	Derive oracles in excel manually	9
11	Analyze functional documents	5
12	Derive oracles in excel manually	9
13	Write test cases inputs in excel	8
14	Derive oracles in excel manually	9
15	Analyze functional documents	5
16	Derive oracles in excel manually	9
17	Write test cases inputs in excel	8
18	Analyze functional documents	5
19	Write test cases inputs in excel	8
20	Derive oracles in excel manually	9

21	Analyze functional documents	5
22	Derive oracles in excel manually	9
23	Write test cases inputs in excel	8
24	Derive oracles in excel manually	9
25	Analyze functional documents	5
26	Derive oracles in excel manually	9
27	Write test cases inputs in excel	8
28	Analyze functional documents	5
29	Write test cases inputs in excel	8
30	Derive oracles in excel manually	9
31	Analyze functional documents	5
32	Derive oracles in excel manually	9
33	Write test cases inputs in excel	8
34	Derive oracles in excel manually	9
35	Analyze functional documents	5
36	Derive oracles in excel manually	9
37	Write test cases inputs in excel	8
38	Analyze functional documents	5
39	Write test cases inputs in excel	8
40	Derive oracles in excel manually	9
41	Create/modify aspects	11
42	Copy Test cases	23
43	Run test cases and generate reports	12
44	Analyze test results	14
45	Analyze coverage of test cases	15
46	write test cases	21
47	Run test cases and generate reports	12
48	Analyze test results	14
49	Modify oracles of test cases in testing tool	17
50	Analyze functional documents	5
51	Modify oracles of test cases in testing tool	17
52	Analyze functional documents	5
53	Modify oracles of test cases in testing tool	17
54	Create/modify aspects	11
55	Modify test cases in testing tool	18
56	Run test cases and generate reports	12
57	Analyze test results	14
58	Analyze coverage of test cases	15
59	Modify test cases in excel	22
60	Documentation	24
61	Analyze functional documents	5

62	Documentation	24
63	Update review checklists	25
64	Archive	26
65	Close task	27

Table 4.2: Task Process of an Average-Productivity Programmer

4.1.2 Modeling the Task Process as a Markov Chain

We use Markov chains to model the task process of a programmer. Markov chains are used in different domains on numerous applications for modeling and analysis of a stochastic (random) process [199]. Typically, a stochastic process representing a system moves randomly between various system states over time. The stochastic process evolves by taking various alternate paths (sequence of states) for the given similar starting point/initial conditions, as happens in task processes. Markov chains model the evolution of such stochastic processes using system states and transitions between the states.

A task process of a task executed by a programmer corresponds to one Markov chain. In a Markov chain model of a task process, each state represents some step, and transition probabilities represent the probability of executing the next step.

After identifying the sequence of steps used by a programmer for executing each task we convert it to a Markov chain. In order to compare Markov chains, we need to have the same size and state space for all Markov chains. For this, we simply added the missing steps as dummy steps in the table of the task - these steps were visited 0 times in the task process. With this, we can covert the task process into a state transition matrix - in this example, it will be a 27x27 matrix. In the matrix, each cell (i, j) has an integer - representing the number of time the transition from state i to j took place. The transition matrix of the task process described above is given in Table 4.3.

Step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1		1																									
2				1																							
3																											
4					1																						
5								4	6		1						2							1			
6								1																			
7																											
8					4				7																		
9					6			6			1																
10																											
11						1												1					1				
12														3													
13																											
14															2		1										
15																					1	1					
16																											
17					2						1																
18												1															
19																											
20																											
21												1															
22																								1			
23												1															
24					1																				1		
25																										1	
26																											1
27																											

Table 4.3: State transition table of the task process of average-productivity programmer

From this state transition table, we can also determine the probabilities of transition from state i to state j - this is simply the number of times transition from state i to state j takes place divided by the total number of transitions from state i. With the total number of states and the probabilities, the Markov chain can be constructed. The chain for the process in the Table 4.3 is given in Figure 4.1.

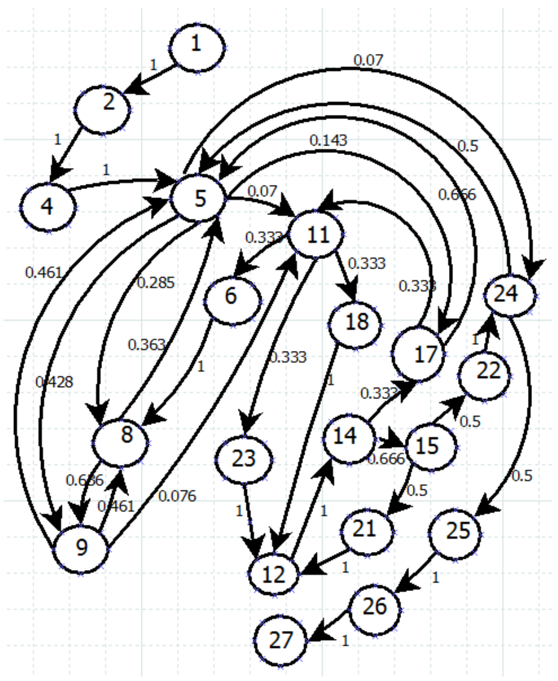


Figure 4.1: A Sample Task Process of Average-Productivity Programmer Modelled as Markov Chain

Similarly, we model all the tasks for average-productivity programmers. And we do the same for all the tasks for high-productivity programmers. As an illustration, the task process table of a task process for a high-productivity programmer is given in Table 4.4. As we can see, this process uses far fewer steps than the task process of an average-productivity programmer shown above, and it has many steps that have not been used.

Sl. No	**Step Name**	**Step. No**
1	Analyze task request	1
2	setup test environment using macros	3
3	Load files	4

4	Analyze functional documents	5
5	compare functional documents with model	6
6	copy information to excel	7
7	Derive oracles in excel using excel formulas	10
8	Analyze functional documents	5
9	Derive oracles in excel using excel formulas	10
10	write test cases inputs in excel	8
11	create/modify aspects in testing tool	11
12	copy test cases	23
13	Run test cases and generate reports	12
14	Analyze test results	14
15	understand code behavior	16
16	Analyze functional documents	5
17	understand code behavior	16
18	modify excel formulas in excel	19
19	copy test cases	23
20	Run test cases and generate reports	12
21	Analyze test results	14
22	analyze coverage of test cases	15
23	documentation	24
24	update review checklists	25
25	Archive	26
26	Close task	27

Table 4.4: Task Process of a High-Productivity Programmer

Again, this table can be converted to a 27x27 state transition matrix, which is shown in Table 4.5 below. The Markov chain diagram for the process is shown in Figure 4.2.

Step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1			1																								
2																											
3				1																							
4					1																						
5						1				1						1											
6							1																				
7										1																	
8											1																
9																											
10					1			1																			
11																							1				
12														2													
13																											
14															1	1											
15																								1			
16					1														1								
17																											
18																											
19																							1				
20																											
21																											
22																											
23												2															
24																									1		
25																										1	
26																											1
27																											

Table 4.5: State transition table of the task process of high-productivity programmer

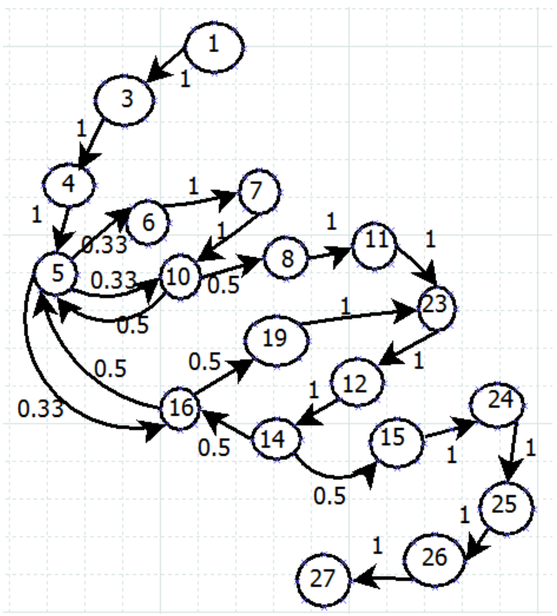


Figure 4.2: A Sample Task Process of a High-productivity Programmer Modelled as Markov Chain

It is evident that programmers use various steps for executing a task and even revisited previous steps such that each step helped move forward toward completion of the task. Though the starting state in both the task processes remained same, the evolutions of these task processes were different from each other. It can also be seen that some of the steps used by a programmer were not observed in the task processes of other programmers and vice versa. Similarly, the total number of steps used by one programmer was different from the number of steps used by another programmer. Also, the order of execution of the steps in a task process was different from one programmer to another programmer.

4.1.3 Difference between Task Processes as Distance between Markov Chains

To study the impact of the difference between task processes, we model the difference between two task processes as the distance between the Markov Chains of those task processes. The dissimilarity between two task processes increases as the distance increases. The distance between two Markov Chains can be computed easily by their state transition matrices (STM), which capture the probabilities of transitions between any two states of a Markov chain.

Let A and B are two matrices of size n*m. Let a_{ij} represents an element in matrix A at the i^{th} row and j^{th} column. Similarly, let b_{ij} represents an element in matrix B at the i^{th} row and j^{th} column. The distance between A and B are computed in many ways, some of them are given below.

$$L_1 = \sum_{i=1}^{n} \sum_{j=1}^{m} |a_{ij} - b_{ij}|$$

or

$$L_2 = \sqrt[2]{\sum_{i=1}^{n} \sum_{j=1}^{m} (a_{ij} - b_{ij})^2}$$

or

$$L_p = \sqrt[p]{\sum_{i=1}^{n} \sum_{j=1}^{m} (a_{ij} - b_{ij})^p}$$

L_1 is called as Sum of Absolute Differences (SAD) between two matrices and is often called as Manhattan distance. L_2 is popularly known as Euclidean distance and is also known as Sum of Squared Differences (SSD). As can be seen, both L_1 andL_2 would be zero between same task processes. In this work, we used Sum of Absolute Difference (L_1 distance) as a distance measure to compute the similarity between two state transition matrices derived from the Markov chain models of task processes.

4.2 Analyzing Task Processes

4.2.1 Research Questions

We have the task processes, as extracted from the video, of each task of each programmer in the study. We also have their Markov chains. We also have the grouping of programmers in the two groups. Using these we can analyze the task processes of programming in the two groups and differences between them. We studied the task processes of high- and average-productivity programmers to investigate the following research questions:

RQ1: Do programmers use similar task process while executing similar type of tasks?

RQ2: How similar are the task processes of high- and average-productivity programmers

The reason behind RQ1 is obvious - we want to study if programmers use similar task processes for similar tasks, and whether the level of similarity is different for high-/ average-productivity programmers. RQ2 focuses on comparing the task processes of programmers within each group (high or average productivity) and across groups to understand whether programmers in a group use similar task processes or not, and whether or not the task processes of high-productivity programmers are different from the task processes of average-productivity programmers.

4.2.2 Study / Experimental Setup

Projects: Three model-based testing projects were selected for this study ensuring that all the three projects belong to the same domain (embedded automotive), test similar functionality, follow similar organizational processes, use same tools and run for a long time. programmers in these teams performed only testing.

Programmers: Programmers from each selected project were chosen in such a way that they had a same educational background, received similar training and had one or two years of model-based unit-testing experience in the projects. A total of 18 programmers participated in this study.

Tasks: Each programmer executed two tasks. Therefore, a total of 36 task processes were modeled as a network of states and analyzed using the concept of Markov chains.

The software size of the tasks executed by programmers was also noted. As a part of organizational practices, the project managers calculated the software sizes of tasks even before allocating the tasks to programmers using the size measure "Adjusted testable requirements (ATR)" [20]. ATR requires first splitting the requirements to low-level requirements such that each low-level requirement is testable and then the complexity of each requirement is measured. Finally, ATR is computed as a function of all weighted low-level requirements where weight of each low-level requirement represents the complexity of the respective requirement.

Like before, input from Project Managers and past productivity data was used to group the programmers as either high-productivity or average-productivity. This was also verified using actual data. We observed that the productivity of high-productivity programmers was 2 to 3 times than that of average-productivity programmers.

4.2.3 Similarity in Task Processes of Programmers (RQ1)

Each of the 18 programmers in this study executed two unit testing tasks. Therefore, a total of 36 task processes were modeled as Markov chains. Among these 36 Markov models of task processes, 12 were from project 1, 10 were from project 2 and 14 task processes were from project 3. The distance between every two task processes in a project was computed as discussed above. The distances between all the task processes in projects 1, 2 and 3 are shown using a box plot in Figure 4.3.

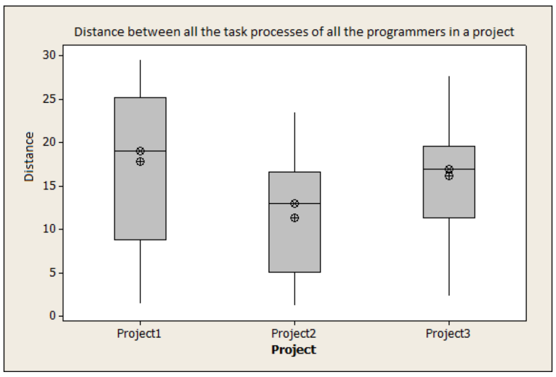


Figure 4.3: Distance between the Task Processes of all Programmers

The mean and standard deviation of the distances between all the task processes in projects 1, 2 and 3 are 17.8 +/ 8.37, 11.3 +/ 6.3 and 16.1 +/ 5.877, respectively. In each project, it is clearly evident from Figure 4.3 that there exist multiple ways of executing a task.

The distance between the task processes of tasks executed by each programmer is shown using a box plot in Figure 4.4. (Figure 4.3 is different from Figure 4.4 as Figure 4.3 shows distance between all the task processes of all the programmers in a project, whereas Figure 4.4 illustrates the distance between the task processes of two tasks executed by a programmer.) The mean and standard deviations of the distance between the task processes of a programmer in projects 1, 2 and 3 are 4.24 +/ 2.4, 4.45 +/ 2.05 and 5.87+/2.79, respectively. As we can see, these are significantly lesser than the distance between the task processes of all the programmers in a project shown in Figure 4.3. We can, therefore, say though programmers do not use exactly the same task process for similar tasks, each programmer uses somewhat similar task processes for executing similar tasks, (and different programmers use different task processes). We can call this as the personal task process (PTP). While PTP varies something for different instances of a task, they are quite similar.

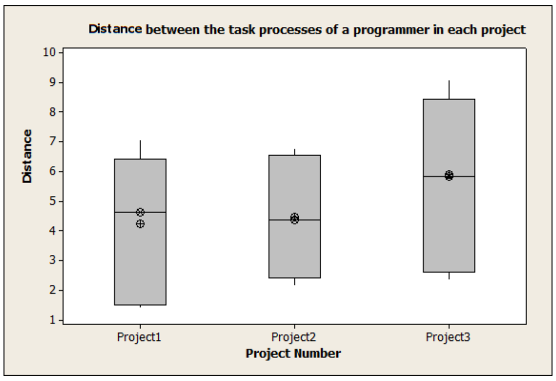


Figure 4.4: Distance between the Task Processes of a Programmer

The distance between the task processes of tasks executed by a programmer of each group (high and average productive) in each project is shown using a box plot in Figure 4.5. (Figure 4.4 is different from figure 4.5 as figure 4.4 shows difference in PTP's of a programmer in each project, whereas figure 4.5 shows difference in PTP's of a programmer in each group.) It is evident from Figure 4.5 that the mean/median of distances between the task processes of a high-productivity programmer is significantly lesser than the mean/median of distances between the task processes of an average-productivity programmer. We can therefore say that the task processes used by a high-productivity programmer's for similar tasks are much closer to each other, than the task processes used by average-productivity programmer's. In other words, we can say that high-productivity programmers have evolved and use relatively stable task process for executing similar tasks, while the average-productivity programmers still seem to be trying variations. This is an important observation regarding how the two groups of programmers execute tasks.

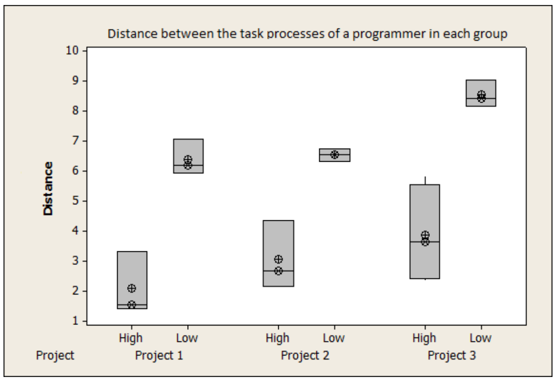


Figure 4.5: Distance between the Task processes of a Programmer in Each Group

4.2.4 Differences between High- and Average-Productive Programmers (RQ2)

Distance between all the task processes of all programmers in a group in a project is shown using a box plot in Figure 4.6. (Figure 4.5 is different from Figure 4.6 as Figure 4.5 shows the distance between PTP's of a programmer in each group, whereas Figure 4.6 shows the distance between the task processes of all programmers in each group.) It is evident from Figure 4.6 that the mean/median of distances between the task processes of high-productivity programmers is much lesser than the mean/median of distances between the task processes of average-productivity programmers. From this we can say that the task processes of programmers in a high-productivity group are more similar to each other, and the task processes of programmers in average-productive group vary from each other.

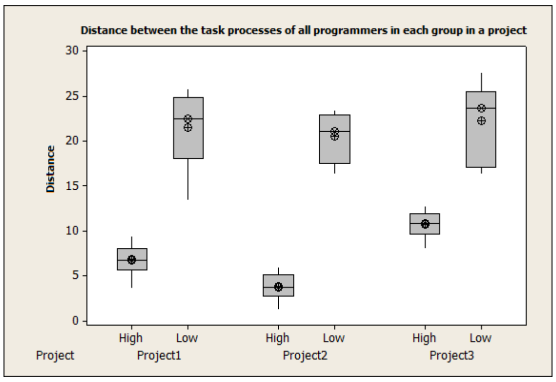


Figure 4.6: Distance between the Task Processes of all Programmers in a Group

The distance between the task processes of high-productivity programmers and the task processes of average-productivity programmers in each project is shown using a box plot in Figure 4.7. As we can see, the distance between task processes of average-productivity and high-productivity programmers is quite large indicating that the task processes of an average-productivity programmer are not only different from the task processes of other average-productivity programmers but also different from the task processes of a high-productivity programmer.

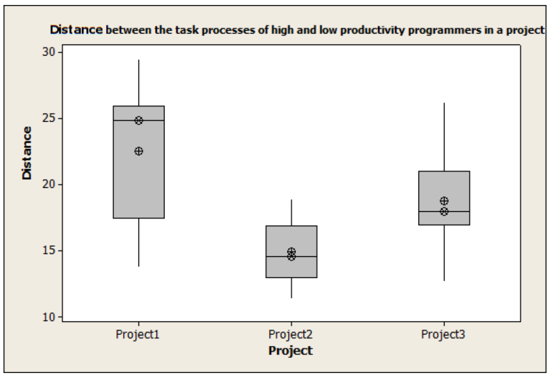


Figure 4.7: Distance between the Task Processes of High- and Average- Productivity Programmers

We also identified differences between the task processes of high- and average-productivity programmers. These differences are grouped into three categories: steps that are present in high-productivity programmers but missing in average-productivity programmers, steps that are most frequently visited by average-productivity programmers but less visited by high-productivity programmers and finally the differences in the captured attributes of steps between high- and average-productivity programmers.

The observed differences between the task processes of high- and average-productivity programmers in model-based unit testing tasks are given in Table 4.6. (These are the differences of average-productivity programmers with respect to high-productivity programmers.)

Missing steps
Automation in test project setup
Moving required information from various documents to a single document
Using excel formulas to calculate expected outputs of test cases
Comparing design/requirements with code-under-test
Selecting signals/variables for analyzing test results and code behavior
Steps that are often revisited by average-productivity programmers
Analyzing design document
Changing the expected output values of test cases
Creation/Modification of aspects (equivalent classes) of input variables
Adding or changing documentation
Executing test cases
Attributes of steps
More number of test cases by average-productivity programmers
Less number of test cases before the first execution
Less time in analyzing the design document during first iteration
More time on test project setup
More time on generating test cases overall the iterations

Table 4.6: Differences in Task Processes between High- and Average-productivity Programmers

Missing steps are identified by using the state transition tables of high- and average-productivity programmers. Each row in the state transition table contains the number of times a transition happened from a step to other steps. If a step is missing in the task process of a programmer, then the number of transitions in that corresponding row will be zero.

Steps that are often revisited by average-productivity programmers are also computed using the state transition tables of high- and average-productivity programmers. The elements in each column of a state transition table correspond to the number of times a transition happened to a step from other steps. We first compute the sum of transitions to each step in a task process by adding all the elements in each column of the respective state transition table. We then identify steps in each task process where a programmer had visited more often than other steps. The identified steps are grouped based on the high- and average-productivity programmers. Finally, the steps are compared across the groups to identify the steps that are often visited by average-productivity programmers but least visited by high-productivity programmers and vice-versa.

As mentioned in the earlier chapters, we have also captured a few attributes of steps while analyzing the task videos of programmers. These attributes were compared between high- and average-productivity programmers based on the mean/median of the captured data to identify the differences between high- and average-productivity programmers.

4.3 Summary

We studied the task processes of high- and average-productivity programmers to investigate how task processes of these two groups of programmers vary. We first modeled each task process as a network of states and then analyzed the task processes using the concept of Markov chains. We compared the task processes of programmers within each group (high/ average productive) and across the other group by comparing the state transition matrices derived from their respective Markov chains. Our study shows that programmers use a similar task process for executing similar tasks and we also observed that task processes of high-productivity programmers are similar to each other, task processes of average-productivity programmers vary with each other, and task processes of high-productivity programmers are different from the task processes of average-productivity programmers. Our next step is to study the effect of transferring the productive task processes to average-productivity programmers on the productivity of programmers.

Chapter 5

Impact of Transferring Task Processes of High-Productivity Programmers to Average-Productivity Programmers

As we have seen, the task processes used by high-productivity programmers to execute an assigned task is often different than the ones used by average-productivity programmers. In our experiments, both sets of programmers (high-productivity and average-productivity) whose tasks we studied are peers and from the same group with similar background, training, and experience. A natural question then arises - can the average-productivity programmers learn from the peers who have a higher productivity and become more productive. This question, in many ways, is the driving motivation behind the study of task processes - that task processes impact productivity and so by improving the task processes, productivity can be improved. And for improving the task process, we can simply learn from the more productive peers - after all, despite same background and training, they have evolved their task processes which seem to lead to higher productivity.

In this chapter, we report results of our efforts to transfer the task processes of high-productivity programmers to the average-productivity programmers in an attempt to improve their productivity. We conducted this study on two projects (treatment projects). In one project, we directly trained the average-productivity programmers on the differences in their task processes with the task processes of high-productivity programmers. In the other project, we did not conduct training, but informed the project manager about the differences, and they further informed the team members. We compare the results with a baseline project, in which no study of differences between programmers was done. All these are similar projects at Robert Bosch Engineering & Business Solutions Private Ltd, a CMMi level 5 software company. The study suggests that it is possible to improve the productivity of average-productivity programmers by transferring task processes of high-productivity programmers to them, and the effectiveness of this transfer determines the productivity benefit that accrues.

5.1 Experiment Setup

To study the impact of transferred task processes on the productivity of average-productivity programmers, we considered three similar model-based unit testing projects. We considered one project as a baseline project and other two projects as treatment projects for comparing the results obtained through our approach. The experimental design is shown in Figure 5.1.

We conducted this study for more than a year at Robert Bosch Engineering & Business Solutions Ltd, a CMMi Level 5 software company, India. Some of the relevant aspects of this experiment are:

Projects: We selected three similar model-based unit testing projects from the same domain and using the same platform. These projects perform testing of similar functionality in embedded automotive domain and use same tools/technology and overall software process.

Programmers: The programmers selected in these projects had a similar educational background, training and work experience. Programmers in these projects have

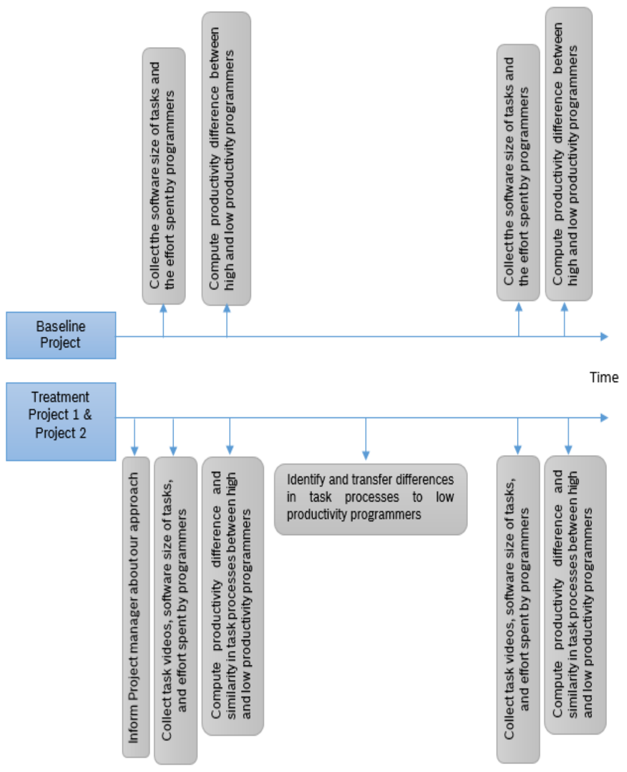

Baseline Project
Collect the software size of tasks and the effort spent by programmers
Compute productivity difference between high and low productivity programmers
Collect the software size of tasks and the effort spent by programmers
Compute productivity difference between high and low productivity programmers
Time
Treatment Project 1 & Project 2
Inform Project manager about our approach
Collect task videos, software size of tasks, and effort spent by programmers
Compute productivity difference and similarity in task processes between high and low productivity programmers
Identify and transfer differences in task processes to low productivity programmers
Collect task videos, software size of tasks, and effort spent by programmers
Compute productivity difference and similarity in task processes between high and low productivity programmers

Figure 5.1: Experimental Setup for Studying the Impact of Transferring Task Processes to Average-Productivity Programmers

1-2 years of experience. A total of 24 programmers from these three projects were considered for this study.

Tasks: Testing tasks in model-based software development were assigned to programmers in this study. Each programmer executed at least two testing tasks at the start and end of the study.

Data collection: For treatments projects, data was collected about task processes as described earlier through analysis of their screen videos. The size data was captured for all tasks separately. In the baseline project, we did not inform the project manager and programmers about our approach as we wanted the baseline project to run normally. We also obtained the effort spent by programmers for executing assigned tasks and verified the effort spent with each programmer. For the baseline project we used the weekly activity report for the effort, while for treatment projects we used the task videos.

We also collected the software size of tasks assigned to programmers. As mentioned, we did not introduce any new size measure specifically for this study but considered the same size measure followed in the organization. “Adjusted testable requirements (ATR)” is the size measure followed in the organization for a very long time. ATR is a combination of the size measures “testable requirements” [20] and “function points.” All the project managers in this study used the same size measure and estimated the software size of tasks assigned to programmers as a part of scheduling and task allocation process.

We studied three projects - two treatment projects and one baseline project. All the projects were studied over a period of about ten months. We studied the productivity of two groups in the “start” of the study, and then we studied the productivities “after” the transfer was done - this was done after the programmers have been independently executed tasks for about six months after the transfer was done. The number of programmers in each project in each of the two groups at the start of the project and after the transfer is shown in Table 5.1.

	Treatment Project1		**Treatment Project2**		**Baseline Project**	
	High Prod. Group	**Avg Prod. Group**	**High Prod. Group**	**Avg Prod. Group**	**High Prod. Group**	**Avg Prod. Group**
No of Programmers at the start	5	7	3	3	3	3
No of Programmers after the transfer	5	6	2	3	3	3

Table 5.1: Number of programmers in the two groups in treatment and baseline projects

(The number of programmers in the two treatment projects has declined by one due to the programmers leaving- one programmer in the average productivity group in the treatment project 1 and one programmer in the high productivity group in the treatment project 2.) The baseline project is included as it provides data on productivity improvement that may take place simply due to experience in executing similar tasks over the time that elapses between the start of the study and when the second measurements are done - the two being about ten months apart. For this project, we collected effort and size for the tasks. In the baseline project, we neither informed the programmers nor project managers about our study, and we did not collect any task videos from the programmers.

In the treatment projects, we collected task videos from programmers at the start of the study. From these videos we did the exercise done before - obtained the task-processes of the two groups and then determined the difference between the processes of the two groups. We then identified the key differences between the task processes of high- and average-productivity programmers which we (including the project managers and programmers) believed were the main cause of the productivity difference.

To transfer the identified differences from high-productivity programmers to average-productivity programmers, we followed two different approaches in the two projects. In the first one (treatment project 1), we had a discussion with the project manager on how we can transfer them to average-productivity programmers. Then we first informed the programmers in the team about the differences between the task processes of high- and average-productivity programmers, and made a presentation explaining how high-productivity programmers executed their tasks. This was followed by the average-productivity programmers executing a couple of tasks together with high-productivity programmers.

In the second treatment project, the transfer was done informally. The project manager was briefed about the study and the differences. He then agreed to share with the team the differences in a suitable manner. No training was conducted for average-productivity programmers.

5.2 Analysis and Results

For all the three projects (the two treatment projects as well as the baseline project) we determined the productivity of the two groups of programmers at the start of the study. We did a similar exercise six months after the transfer of process was done in the two treatment projects, i.e. - computed the productivity of two groups for each of the projects. For computing the average productivity of a group, we computed the productivity of each programmer, and took the average. The average productivity of various groups at the start and after the transfer is show in Table 5.2 (Productivity is in ATRs tested per hour.)

As we can see, the productivity of average-productivity programmers improved from 0.8 to 1.9 in the treatment project 1 where we transferred the task processes to average-productivity programmers through training. And the average productivity improved from 0.69 to 1.05 in the treatment project 2 where the transfer of task processes was done implicitly through team meetings. So there was an improvement of 135% and 52% in the productivity of average-productivity programmers in the treatment projects. In the baseline project, during the same period, the productivity of average-productivity programmers also improved from 0.85 to 0.98, that is, an improvement of 15% - this improvement in productivity of average-productivity programmers probably represents the effect of natural effort that average-productivity programmers must be putting to improve by learning from team members.

	Treatment Project1		**Treatment Project2**		**Baseline Project**	
	High Prod. Group	**Avg Prod. Group**	**High Prod. Group**	**Avg Prod. Group**	**High Prod. Group**	**Avg Prod. Group**
Initial Productivity	2.2	0.8	1.8	0.69	1.9	0.85
Final Productivity after transfer	2.5	1.9	1.95	1.05	1.98	0.98

Table 5.2: The productivity of each group in treatment and baseline projects

The productivity of high-productivity programmers also improved marginally in all the projects. Interestingly, the improvement is more in the treatment projects where they were involved in training the average-productivity programmers, when compared with the baseline project.

At the start of this experiment, the ratio of productivity of high-productivity programmers and average-productivity programmers was 2.75 and 2.6 in the treatment projects, and 2.25 in the baseline project. After the experiment, this ratio reduced to 1.3 and 1.85 in the treatment projects, while in the baseline project, this ratio reduced to 2 - a much lesser improvement. In other words, the difference in productivities between high- and average-productivity programmers decreased by 57% and 19% in the treatment projects, while in the baseline project it was 5% only.

The individual productivity of average-productivity programmers "before" and "after" in the three projects is shown in Table 5.3. As we can see, the productivity of all of them improved. However, the improvement is largest and most consistent in the treatment project 1, where the productivity of all have improved by more than 2X (except programmer 4 where it is about 2X).

	Treatment Project1		**Treatment Project2**		**Baseline Project**	
	Before	**After**	**Before**	**After**	**Before**	**After**
Programmer 1	1.11	2.47	0.47	0.71	1.05	1.2
Programmer 2	0.91	2.175	0.77	1.44	0.725	0.84
Programmer 3	1.01	2.31	0.82	1.29	0.795	0.9
Programmer 4	0.49	0.97				
Programmer 5	0.83	1.855				
Programmer 6	0.65	1.585				

Table 5.3: Individual productivity of average productivity programmers

We also studied the similarity between the task processes in the treatment projects. If the similarity between the task process of high-productivity programmers and the task process of average-productivity programmers increases (i.e. the distance between them decreases), we can safely conclude that the improvement in productivity is due to the improvement in task process. For similarity between the task processes, as before we derived the state transition matrix for each task process, modeled each task process as a Markov chain, and computed the distance between the state transition matrices as the similarity measure between two task processes. We computed the distance between the task processes of high- and average-productivity programmers at the beginning and end of the study for the treatment projects. (We did not do this in the baseline project as they were not made aware that such a study was going on.) The similarity between the task processes of high- and average-productivity programmers in the two treatment projects is shown in Figure 5.2.

As we can see, the similarity between the task processes of high- and average-productivity programmers improved substantially - the distance between them reduced by 44% and 13% in the two treatment projects.

We conducted direct interviews with four average-productivity programmers of the treatment project1 (about a year after the training sessions). We met each of these four average-productivity programmers separately and reminded them of the missing steps and steps that were being revisited often (as discussed in the earlier chapter) that were discussed in the training. The questions that we asked in the interview is given

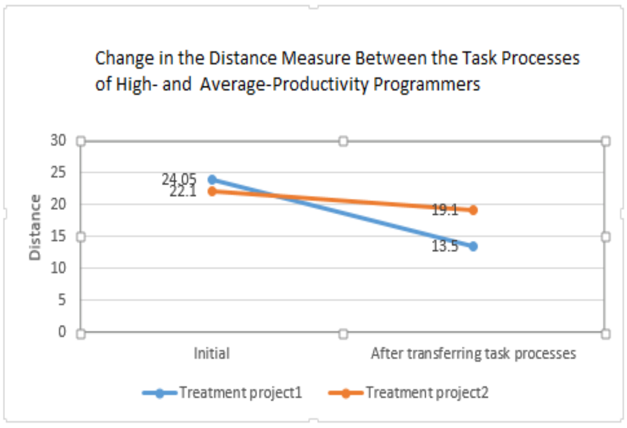


Figure 5.2: Change in the Distance between the Task Processes due to the Transfer of Task Processes

in Table 5.4.

Table 5.4: Interview Questions to Average-productivity Programmers

Questions
Was the training session helpful for improving productivity?
Why did you think your productivity improved?
Did you feel the execution of tasks changed due to the learning you had?
Which of these steps did you adopt in session?
Any feedback that you would like to share?

All the four average-productivity programmers in the treatment project1 found the session helpful for improving productivity. Two of these average-productivity programmers were surprised to see a large variation in the generated test cases and the number of test executions by the programmers, and feel that the data had helped them to introspect and modify their process for execution of tasks.

All the programmers think that they had improved their productivity. Three out of four average-productivity programmers feel that they are now executing their tasks quickly with ease when compared to earlier. Though the fourth programmer also feels the same, he opinioned that further improving the way of generating test inputs can boost more improvement in productivity.

For our question “which of these steps did you adopt from the session?”, all the four programmers told us that they are now using excel formulas for deriving test case outputs, using macros in the step test project setup, spending time to analyze functional documents and code at the beginning, comparing signals for analyzing the behavior of code, generating less number of test cases & test executions when compared to earlier, and copying the required information from documents to a common place.

We have also verified the task processes of average-productivity programmers after the training. We found that all average-productivity programmers in treatment project1 are using all the missing steps. The average-productivity programmers in treatment project2 are not using the step “copy required information to a common place” (This may be due to the effectiveness of training in the treatment project2.)

Two average-productivity programmers opinioned that it is required to revisit the step “analyze functional document” multiple times in the course of task execution. One average-productivity programmer told us that it was difficult to document the generated test cases at the end in one shot. He feels that it is always better to perform the documentation of test cases intermittently throughout the task execution and he finds it okay to change the documentation for any change in the generated test case inputs.

We have also asked them to share any other feedback. While some of them are more general, some of the ones pertinent to this study are:

- Though “Lessons learned and best practices” meetings are happening regularly in the project, these meetings do not discuss in depth about steps (sub tasks) in the task, etc. for identifying the best practices of task execution. Project manager should conduct “lessons learnt and best practices” meetings at regular intervals asking each programmer how he/she execute various steps (sub tasks) in a task and then stimulating a thought provoking discussion among the team to check whether or not any best steps exists for adoption by all programmers.

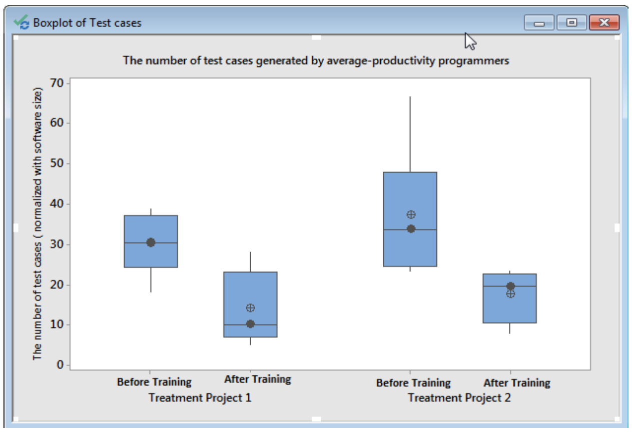


Figure 5.3: The Number of Test cases Generated by Average-productivity Programmers Before and After the Training in Treatment Projects

- The productivity gains that can be achieved by identifying the best ways of executing a task with in a team could be less. This should be done across the teams and the organizations for maximum productivity gains.

- You may directly discuss with each programmer to understand how he/she executes a task. Otherwise, sit with each programmer while he/she is executing a task and understand the task better. Video recording of tasks may be avoided. I hope my videos were not shared with anyone without my consent.

We could not get in touch with the average-productivity programmers from the treatment project2. However, we have asked the project manager of the treatment project2 about any feedback that he would have received formally or informally from his team members. The project manager felt that the study helped to identify areas of improvement and actually improved the productivity of his team members. However, he also feels that the study is little risky by itself and if not executed properly in the live projects then it may lead to a large disturbance in the cooperation between the project manager and team members.

We have also noted the number of test executions and the generated test cases by average-productivity programmers after the completion of training in treatment1 and treatment2 projects. As it can be seen from the Figure 5.3, the mean/median of

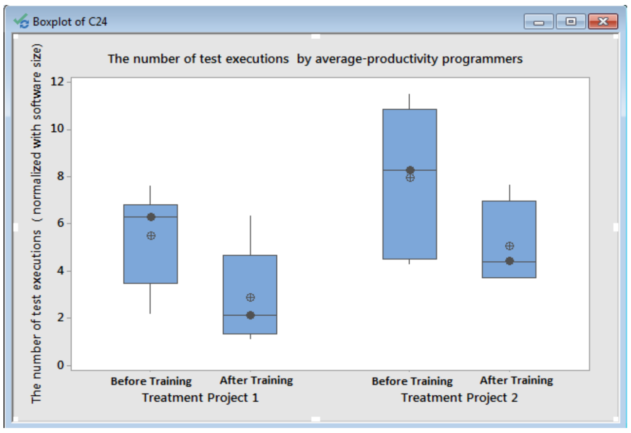


Figure 5.4: The Number of Test Executions by Average-productivity Programmers Before and After the Training in Treatment Projects

generated test cases (normalized with the size of tasks) by average-productivity programmers after the training has improved in both projects. Similarly, the number of test executions after the training also shows an improvement in both the projects in Figure 5.4.

The remarkable improvement in productivity and improvement in similarity with task processes of high-productivity programmers clearly suggests that productivity of average-productivity programmers can be improved by improving their task processes. And for significant improvement, they need to just follow the task processes of those in their peer groups who have high productivity.

The study also suggests that for transferring the processes of high-productivity programmers, an active approach is much more effective. Just leaving the programmers to learn from peers through informal interactions is not as effective as a structured and planned transfer through training by peers.

Overall, the study further confirms that task processes used by programmers have a strong influence on programmer productivity. And the productivity of average-productivity programmers can be improved significantly and brought closer to higher productivity peers, by systematically identifying the key differences between how high- and average-productivity programmers execute a task, and transferring the processes of high-productivity programmers to average-productivity programmers through planned and properly conducted training.

5.3 Summary

Improving software productivity is one of the most important goals of software development companies. One way to improve the software productivity is by improving the productivity of programmers. We know that some programmers are much more productive than other programmers in a project, and so productivity can be improved by improving the productivity of average-productivity programmers. We hypothesized that one of the main reasons why high-productivity programmers are more productive is due to their use of more effective task processes to execute the tasks they are assigned. We further hypothesized that it should be possible to get the benefits of the effective task processes of high-productivity programmers by transferring them to the average-productivity programmers.

In the studies reported in earlier chapters, we observed how high-productivity programmers execute their tasks when compared with average-productivity programmers (the organization of various steps used by a programmer for executing his/her assigned task is called as a task process). We have reported differences between the task processes of the two groups. In the study reported in this chapter, we discuss the impact of transferring task processes of high-productivity programmers to average-productivity programmers in the project.

We conducted this study on two live model-based unit testing projects (called the treatment projects) at Robert Bosch Engineering and Business Solutions Private Limited, a CMMi Level 5 software company. In one treatment project, we transferred the task processes of high-productivity programmers to average-productivity programmers by training the average-productivity programmers. In other treatment project, we only informed the team about the differences through the project manager. For comparison purposes, we studied another similar project (called the baseline project). We found that the productivity of average-productivity programmers improved by about 135%

and 52% in the treatment projects. The difference in productivity between the high- and average-productivity programmers decreased by 57% and 19%. The corresponding figures in the baseline project were much lower.

To understand whether or not the productivity improvement is due to transfer of task processes to average-productivity programmers, we also compared the distance between the task processes of high- and average-productivity programmers by modeling each task process as a Markov chain. We found an improvement in the similarity between the task processes of high- and average-productivity programmers by 44% and 13% in the treatment projects.

The study shows that we can improve the productivity of average-productivity programmers substantially by identifying the key differences between task processes of high- and average-productivity peer programmers, and training the average-productivity programmers on the key differences to transfer the processes of high-productivity programmers. The study also suggests that the level of improvement depends on the effectiveness of this transfer - direct transfer through training shows better results than the indirect and informal transfer within the team.

Chapter 6

Other Studies For Improving Programmer Productivity

In this chapter, we discuss some other studies that we had conducted on improving programmer productivity. In the first study, we report results of a study we did to understand the impact of the trainer on the trainee - whether the productivity of the trained programmer is influenced by the productivity of the trainer. In the second study, we study the impact of taking some steps to counter the Parkinson's law (i.e. the work expands to fill the available time) and what impact they have on programmer productivity. In the third study, we study how model-based development impacts programmer productivity in enhancement tasks, as compared to in a traditional development project.

6.1 Using High-Productivity Programmers for Training New Programmers Improves Team Productivity

Attrition of team members in a project demands project manager to use the effort of his/her senior programmers for training new programmers coming into the project. Due to project schedule pressures, often a project manager may not put the best senior programmers for training - instead, use average senior programmers for training new programmers. The assumption is that as training process and material is already defined and available, training by any senior programmer would yield similar results.

6. OTHER STUDIES FOR IMPROVING PROGRAMMER PRODUCTIVITY

We investigate through a case study, the effect of productivity of trainers on the productivity of new programmers and the team by conducting a limited study on a live project at Robert Bosch Engineering and Business Solutions Private Limited, a CMMi Level 5 software company.

Four senior programmers were selected for training new programmers. Two of the trainers were identified as high-productivity programmers and the other two as average-productivity programmers. Both sets of trainers trained two new programmers. After completion of the training and the new programmers working independently for at least six months, the productivity of new programmers, and its impact on team productivity was examined. We found that the programmers trained by high-productivity programmers were 1.75 times more productive than the programmers trained by average-productivity programmers indicating the importance of engaging the better programmers for training new programmers. We also compared the processes used for executions of tasks by new programmers with that of the processes used by their trainers. We found that there is a strong similarity between the two, further strengthening the findings that productivity of the trainers strongly influences the productivity achieved by the new programmers.

6.1.1 Background

The attrition of team members in a software project is common in software companies particularly at junior levels. Attrition results in new programmers regularly getting inducted in long-running projects. New programmers have to be sufficiently trained before they work on actual project tasks independently, as training is seen as the most effective means of knowledge transfer [200, 201]. It is common in software development companies to deploy a dedicated trainer, who is generally another senior programmer from the project, for training a new programmer.

It is the responsibility of a project manager to choose and assign a senior programmer from the project for training a new programmer. Training a new programmer demands a considerable amount of effort by a senior programmer to make the new programmer suitable for executing the project tasks independently. Besides formal training using standard presentations, the senior programmer also spends the effort to review the work of new programmer for finding defects/bugs, clarify doubts and give suggestions on the work executed by new programmers. Clearly, good training is important for proper induction of the new programmers.

However, a project manager's primary responsibility is to ensure timely completion of the activities/tasks according to the schedule and delivery of the project [202]. Therefore, often a project manager may not prefer to spare the best programmers in the team for training a new programmer but choose average programmers from the team for the training task. Project managers often think that training new programmers by any senior programmer will be similar as the focus of training is generally on knowledge transfer and team's processes, and a team generally has standard training documents and presentation material for training new programmers which are used by all trainers. Therefore, often, high-productivity programmers are put on actual work while training responsibility is given to average-productivity programmer to ensure timely completion of the tasks in the projects.

In this work, we study the impact of trainer's productivity on the productivity achieved by the trainee, and the overall team productivity through a case study.

In the case study, we divided senior programmers in the project into two groups - high-productivity and average-productivity, based on their productivity computed on sample tasks and the feedback from the project manager. We selected two senior programmers from each of the two groups for training two new programmers each. That is, two high-productivity programmers trained two new programmers and two average-productivity programmers trained the other two new programmers. The productivity of the new programmers was computed and compared after these programmers had worked independently for six months after the training.

To further confirm the impact of the trainer's productivity, we also studied the similarity between the new programmer's and the trainer's task execution using the methods described earlier.

6. OTHER STUDIES FOR IMPROVING PROGRAMMER PRODUCTIVITY

We conducted this study on a live model-based unit-testing project at Robert Bosch Engineering and Business Solutions Private Ltd, a CMMi Level 5 software company. Our study suggests that it is important that project manager picks high-productivity programmers for training new programmers for improving the productivity of the team.

A lot of work has been done on software engineering education, learning and training [203, 204, 205]. Many studies based on a survey of employees and managers have shown that training has a significant impact on employee's performance [206, 207]. Bartel verified around 3800 employee's personal records of a large company and found that the time spent on training had a significant impact on the performance ratings of employees [208]. Andries conducted an experiment by randomly assigning workers to control and treatment groups and studied their performance before and after the training. He found an improvement of 10% in the performance of workers [209] who had undergone training. This study is different from them regarding both objective (goal) and approach. The existing studies did not differentiate trainers based on high- and average-productivity and study their influence on the productivity of trainees.

Training New programmers:

Knowledge is organized into two types: explicit and tacit [210]. Explicit knowledge can be easily documented and transferred from one individual to others through class room/lecture mode of training. Tacit knowledge, on the other hand, is difficult to document and transferred to others through class room/lecture mode of training as this is carried out in the minds of individuals and is hard to access and express. One way to transfer tacit knowledge is to make new programmer spend a considerable amount of time with senior programmer [211].

Therefore, software development companies use a dedicated senior programmer for training a new programmer till the new programmer is capable of executing his/her tasks independently. So the project manager assigns to a senior programmer from the project the entire responsibility of training a new programmer. The senior programmer acts as the first point of contact for the new programmer for all the project-related information.

In Bosch, training contains three phases: lecture phase, demonstration phase, and clarification phase. Lecture phase is similar to the classroom mode of training/presentations. Senior programmers use the standard approved training material directly from the project repositories for training the new programmers.

Demonstration phase typically starts after the finish of the lecture phase. Demonstration phase may not have any presentation material. Senior programmer explains and shows the way of executing tasks directly on his/her computer. Demonstration phase is more of a pre hands-on training to the new programmer. The new programmer would be observing the senior programmer (an aspect similar to pair programming) while the senior programmer is executing his/her tasks.

Clarification phase starts after the completion of the lecture phase and the demonstration phase. In clarification phase, a senior programmer first starts allocating small tasks of lesser complexity to a new programmer and closely monitors the work done by the new programmer. The senior programmer then reviews the work to find defects, clarify doubts and give suggestions on the work executed by new programmers. Later, the senior programmer starts assigning tasks of increasing complexity to the new programmer and guides him/her throughout the process of execution of tasks.

Project manager keeps track of the entire process and progress of the training. Training finishes after both the project manager and the senior programmer feel that the new programmer has reached a stage of executing the tasks independently. Typically, training would take about one month. We studied the productivity of new programmers after six months from the completion of the training for studying the impact of using high-/average-productivity programmers for training new programmers on the productivity of the new programmers and the team.

6.1.2 Field Study and Data Collection

We conducted a limited study on one live model-based unit-testing project at Robert Bosch Engineering and Business Solutions Private Ltd for studying the impact of using high- and average-productivity programmers for training new programmers on the productivity of new programmers and the team. The project is running for a long time and performs only unit-testing of safety software in embedded automotive domain. The project manager is part of the team for a long time and is well aware of the performance of each member of the team. The setup of our study is given in Figure 6.1.

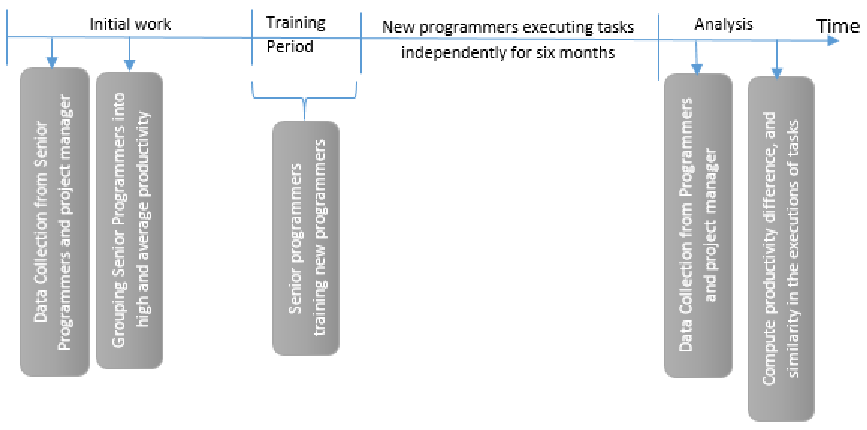

Initial work
Training Period
New programmers executing tasks independently for six months
Analysis
Time
Data Collection from Senior Programmers and project manager
Grouping Senior Programmers into high and average productivity
Senior programmers training new programmers
Data Collection from Programmers and project manager
Compute productivity difference, and similarity in the executions of tasks

Figure 6.1: Setup for Studying the Impact of Using High- and Average-Productivity Programmers for Training New Programmers

We first had a discussion with the project manager for a common understanding about this study. Six senior programmers had a similar educational background, relevant experience and received similar training. We collected feedback from the project manager on the productivity of senior programmers. We also used data from studying their task videos to group them into high and average productivity.

From the group of six senior programmers, we selected two high-productivity programmers and two average-productivity programmers for training four new programmers. The productivity of the trainers (measured as ATR per hour) is given in Table 6.1.

Table 6.1: Productivity of Four Senior Programmers in ATR/hour

	High-productivity programmer	Average-productivity programmer
Trainer 1	1.95	0.80
Trainer 2	1.70	0.67
Mean Productivity	1.82	0.73

All the new programmers chosen for this study had a similar educational background and were recruited directly from college. They had completed similar organizational training (induction and skill improvement training) before they are placed on project-specific training. Two of the four new programmers were trained by high-productivity programmers, and the other two new programmers were trained by average-productivity programmers. The project-specific training lasts around one month.

After completion of the project-specific training and the new programmers working independently for six months, we collected the task videos of at least two model-based unit-testing tasks from all the four new programmers. Project manager's opinion on the productivity of new programmers was also taken. We also compared the task process of the new programmers with those of their trainers, to study the level of similarity between the processes used by the new programmers and their respective trainers.

6.1.3 Analysis and Results

For the new programmers, we determined their productivity after they have been on the project for six months after training. We also studied the task processes they used for completing their unit testing tasks, and how similar they are to the task process used by their trainers. In this section, we give the results of these two.

Productivity of new programmers:

After completion of the training and the new programmers working independently for six months, the productivities of new programmers were computed. The produc-

Table 6.2: Productivity of Four Trainees (New Programmers) in ATR/hour

	Trained by High-Productivity Programmer	Trained by Average-Productivity programmer
New programmer 1	1.10	0.60
New programmer 2	0.95	0.57
Mean Productivity	1.02	0.58

tivity of the four new programmers is given in Table 6.2.

As we can see, the mean productivity of new programmers trained by high-productivity senior programmers was 1.02 ATR/hour, and that of the new programmers trained by average-productivity senior programmers was 0.58 ATR/hour. The productivity of the two trained by high productivity programmers is very similar - so is the productivity of the two trained by average-productivity programmers. However, as we can see, there is a marked difference between the productivity of the two groups - the first set of new programmers is 1.75 times as productive as the second set.

The similarity between task processes: We used the same methodology and tools that we used in our previous studies for assessing the similarity between how the new programmers executed the tasks, and how their trainers executed the task (i.e., how similar were the task processes of new programmers and their trainers).

From the captured videos, we extracted the steps and modeled the execution of a task as a Markov chains as discussed earlier in Chapter 4. Each task executed by a programmer corresponds to one Markov chain. The executions of tasks by programmers were compared by comparing Markov chains of the task processes. Comparison of Markov chains was done by measuring the distance between the two state transition matrices - higher the distance between the state transition matrices of Markov chains, the lesser the similarity between the executions of tasks.

The mean of the distance between the task processes of the different groups is given

Table 6.3: Mean of the Distance between the Task Processes of Trainers and Trainees

	High-Productivity Trainers	Average-Productivity Trainers
New programmers trained by high-productivity programmers	11.88	25.09
New programmers trained by average-productivity programmers	22.55	11.05

in Table 6.3. Higher the distance the lesser the similarity between the executions of tasks.

As we can see, the distance between the task processes of the trainees and their trainers is around 11 for both the trainee sets. That is, the task processes of the trainees are similar to that of their trainers - regardless of whether the trainer was high- or average-productivity. Table 6.3 also shows that the task processes of trainees are very different from the task processes used by trainers of the other group (average of 22 and 25). Overall, this data demonstrates that the task process used by new programmers is strongly influenced by the task process of the trainers. As task processes have a significant impact on the productivity of a programmer, we can also conclude that the new programmers trained by high-productivity programmer's end up being high-productivity programmers as they use the high productivity task processes.

We also collected the qualitative feedback from the project manager on the productivity of new programmers. The project manager also feels that the new programmers trained by high-productivity programmers are more productive than the new programmers trained by average-productivity programmers. Project manager's feedback corroborated well with our results on the productivity of new programmers.

Overall Project Productivity: The new programmers trained by high-productivity programmers are more productive than the programmers trained by average-productivity programmers. The productivity of the new programmers trained by high-productivity programmers was 1.75 times the productivity of new programmers trained by average-productivity programmers. This means the new programmers trained by high-productivity programmers complete their tasks 1.75 times faster than the new programmers trained by average-productivity programmers.

The productivity difference between the high- and average-productivity senior programmers is 1.09 ATR/hour. The productivity difference between the new programmers trained by high- and average-productivity programmers is 0.44 ATR/hour. Even if we assume that in the one month used for training, the trainer spent the time exclusively for training, we can see that the loss in delivery from high-productivity senior programmers during the period of training can be easily covered with the gain in delivery from the new programmers trained by them within about 2 months. After this break-even period, the overall project will benefit by delivering more output because of the additional productivity from the new programmers trained by high-productivity programmers.

This study suggests that it is important that project manager makes an intelligent investment of using high-productivity programmers for training new programmers for long-term benefits of the project. Any decision based on short-term delivery gains may hinder the project progress in the long run.

6.1.4 Summary

Attrition of members in a project often demands project manager to use some effort from his senior programmers for training new programmers coming into the project. Utilizing the effort from best senior programmers for training new members would impact the project schedule and deliveries. Therefore, often a project manager may not put high-productivity senior programmers but average-productivity senior programmers for training new programmers.

In this work, we studied the impact of using high-productivity senior programmers for training on the productivity of the new programmers and the team.

6.1 Using High-Productivity Programmers for Training New Programmers Improves Team Productivity

We conducted a case study on a live model-based unit-testing project at Robert Bosch Engineering and Business Solutions Private Limited.

We found that the productivity of the programmers trained by high-productivity programmers was similar, so was the productivity of the programmers trained by the average-productivity programmers. However, the productivity of the former group of programmers was about 1.75 times the productivity of the latter group, when measured six months after the training.

We also found that there is a strong similarity between the processes used by the new programmer and the process of the trainer. This means that during training, the new programmer effectively learns the task process of the trainer, and tries to use the trainer's task process in his/her tasks. Consequently, the impact of the task process is reflected in their productivity, thereby further confirming that the productivity of the trainers has a significant impact on the productivity of new programmers.

We also found that the loss in delivery from the high-productivity programmer during the period of training could be recovered within about two months by the gains in productivity of the new programmers.

This is a case study conducted on one project, two sets of trainers each heaving two trainers, and two sets of new programmers, each having two. While the results of the study show some outcome and while the outcomes seem consistent with expectations, this study does not provide a statistical basis for a generalized claim. For that many more studies need to be done.

6.2 Countering Parkinson's Law for Improving Programmer Productivity

One possible reason for productivity being lower than what is possible may be due to Parkinson's law, which states that work expands to fill the time available for its completion. In a software project this means that if more than needed time is given to a programmer, the extra time will not be revealed as "free time" on the programmer's weekly activity reports, but will result in the programmer consuming all the allotted time resulting in loss of productivity. A simple approach of allotting less time may not work as there are often small reasons relating to clarifications/coordination that provide the "reason" to a programmer for taking more time for completing a task. Therefore, to counter the effect of Parkinson's law, we adopted a two-pronged approach: (1) allocating 33% less time than the estimated effort for a task and (2) facilitating issue resolution that may impede progress through a 15-min time-boxed daily meeting. We conducted an experiment for about six months in seven software projects in the real environment to study the impact of this approach. We found an improvement of at least 15% in productivity of the programmers compared to their baseline productivity without any degradation in the quality of the programs developed by them.

6.2.1 Background

Typically, a project manager estimates the effort required for a programmer to complete a particular programming task and then uses this effort estimate for estimating the time required. This effort estimate is normally based on the programmer's past (baseline) productivity and the estimated size of the program to be developed. Even though this estimate is based on the programmer's baseline productivity data, it is well known that the baseline data might not reflect the current productivity of the programmer, and so this estimate might be more than the required effort. Furthermore, project managers often consider uncertainties and other constraints and assign more time than is strictly needed.

When a programmer is assigned more time than required following Parkinson's Law, he/she executes the task in such a way that it fills the assigned time. For example, if the programmer realizes that the task could be completed earlier than the assigned time, instead of reporting free time (early delivery with less effort), he may try to fill the free time by self-checking the task multiple times or frequently testing than is normally required.

In some cases, when programmers realize that they were assigned more time than required, programmers may exhibit procrastination. These behaviors lead to reporting more effort than is actually required thereby degrading the programmer productivity. Moreover, in the course of execution of the task, when an uncertainty occurs, the programmer tends to resolve the issues by self-means or through peers rather than getting it resolved with the help of project manager. Thereby, issues related to clarifications/coordination provide programmers "the reason" for taking more time than is actually required.

To counter the effect of Parkinson's law, we took a two-pronged approach: (1) allocating 33% less time than the estimated effort for a task and (2) facilitating issue resolution that may impede progress through a 15-min time-boxed daily meeting. We conducted an experiment for about six months in seven software projects involving over 70 tasks in the real environment to study the impact of this approach. We found an improvement of at least 15% in productivity of the programmers compared to their baseline productivity without any degradation in the quality of the programs developed by them.

6.2.2 Experiment and Data Collection

Subjects: All the programmers with one to four years of experience were included in the seven embedded software development projects.

Objects: Embedded software development (model-based development) tasks having an estimated effort of at least 25 hours were considered in this experiment. The main reason for selecting this criterion of 25 hours was due to the assumption that there might not be any Parkinson's law behavior for small duration tasks.

Task Assignment: Project manager assigned objects to subjects based on the existing project planning and scheduling process.

Variables: Software size of the task, estimated effort, planned effort, the actual effort spent by the programmer for completing the task, baseline productivity of the programmer, and the number of defects identified after completion of the object are the independent variables. Percentage improvement in productivity is the dependent variable. Planned effort assigned to the programmer for completing an object is the control variable.

This experiment was stretched over a period of six months on seven live embedded software projects involving over 70 tasks. Number of tasks for which the data was collected in the projects were: 20, 14, 16, 7, 9, 3, and 3 respectively. To begin with, the project managers of these seven projects were briefed about this experiment. They were requested to allocate an effort of 33% less than the estimated effort and to call for 15-min time-boxed daily meeting to facilitate resolving any issues that impede the task progression.

The project managers were given liberty to change this based on their project scenario. Programmers were informed that extra effort will be available in case they cannot complete the task within the planned effort, to relieve any pressure, and in case there is no Parkinson's law behavior.

We recorded task size, baseline productivity of the programmer, estimated effort, planned effort assigned for the task, actual effort taken by the programmer for completing the task, and the number of defects identified after completion of the task. Software size of the object was estimated in terms of adjustable testable requirements (ATR), which is an internal measure and a variation of the testable requirements metric. The baseline productivity of each programmer was derived from organization database based on the similar type of projects and tasks executed in the past. The estimated effort was calculated based on the software size and the baseline productivity of the programmer. (Estimated effort = Software size /Baseline productivity.) Data for a few

Table 6.4: Data on a few tasks

Trial	Size of the object (ATR)	Baseline Productivity of the subject (size/effort in days)	Estimated Effort to complete object (hours)	Planned Effort allocated to subjects (Hours)	Actual effort taken by subject (hours)	Defects Found
1	2.82	0.8	30	20	21.5	0
2	7.8	0.73	91	61	82.8	0
3	3.95	0.8	42	28	28.8	0
4	9.63	0.85	96	64	89.7	0
5	3.81	0.8	40	27	39.5	0
6	3.56	0.8	38	25	35.2	0
7	3.23	0.8	34	23	31.6	0
8	5.03	0.8	53	36	38.3	0

tasks is shown in Table 6.4. (One day is taken to be 8.5 hrs.)

Planned effort assigned for a task was 33% lower than the estimated effort. The actual effort spent by the programmer to complete the task was captured on a daily basis and the same was compared with the working hours spent in the organization to check for errors in reported effort. Defects found in the tasks executed by programmers was also captured and analyzed to check whether or not there is any impact of this method on the quality of the tasks developed by programmers.

We tried to minimize the confounding issues through following measures: (a) all the objects in this study were of similar type (embedded software tasks on model-based development), (b) software size considers complexity of the object, (c) project manager followed the same estimation criteria as per the organization guidelines to estimate the software size of the object, (d) programmers having one to four years of experience were chosen for the study, and (e) project manager assigned tasks to programmers based on the already planned project scheduling process.

Table 6.5: Productivity improvement in projects

Project	No of tasks	Avg percentage productivity improvement
Project 1	20	11
Project 2	14	28
Project 3	16	34
Project 4	7	61
Project 5	9	56
Project 6	3	11
Project 7	3	16

6.2.3 Analysis and Results

We know that

% Productivity improvement in a task = [(Software size of object/Actual effort taken by subject) - Baseline productivity of subject]*100/ Baseline productivity of subject

This can be simplified to

% improvement = ((Estimated Effort /Actual effort taken) - 1) * 100

We computed the average productivity gain in each of the task. The average productivity gain for each of the project is summarized in the Table 6.5 (in this the average for the project is average gain of all the tasks in the project.) As we can see, significant productivity improvement was observed in all projects, though it was much larger in some of them. The average improvement in productivity over all the tasks in all the projects was around 30%. The detailed data showed that there were six tasks in which a productivity decline was observed, i.e. the actual effort consumed was more than the effort initially estimated, and there were eight tasks in which the improvement was very small and cannot be considered significant. This shows that while the technique worked overall, it cannot be applied blindly to all the tasks.

We did not observe any defects in the tasks executed by programmers and hence no degradation of quality is observed in the tasks executed by programmers. This might be due to the rigorous organizational processes that programmers have to follow. Also, programmers were allowed to take extra time if needed, so there was no pressure to cut corners.

We interviewed some of the project managers and programmers to understand the effectiveness of the study. We found that this experiment has helped the project managers to better understand real productivity of programmers, which in turn helped them to improve effort estimation for future tasks. Project managers did not find programmers stretching themselves to complete the assigned tasks. Some project managers also believed that this method might yield effective results for the tasks estimated with more than 40 hours of effort. Some programmers were of the opinion that the daily meetings helped them to resolve their issues in a timely manner and manage the execution of tasks effectively.

6.2.4 Summary

Software development companies continuously strive for improving productivity. A likely reason for lower productivity than what is possible may be due to the existence of Parkinson's law which states that work expands to fill the available time. Allotting less time for a task may not be a viable solution as there are often small reasons relating to clarifications/coordination that delay completion of a task. To counter the effect of Parkinson's law, we therefore, took a two-pronged approach: (1) allocating 33% less time than the estimated effort for a task and (2) facilitating issue resolution that may impede progress through a 15-min time-boxed daily meeting. We studied the impact of this approach in a CMMi level 5 software company for about six months in seven software projects in the actual field environment. We found at least 15% improvement in productivity of the programmers without any degradation in the quality of the tasks developed by them.

6.3 Effect of Model-Based Software Development on Productivity of Enhancement Tasks

Model-based software development promises to increase productivity by generating executable code automatically from design/models thereby eliminating the manual coding phase. Although new software development projects have yielded high productivity using model-based development, its effect on the productivity of maintaining projects involving enhancement tasks are not well researched. We study the impact of model-based development on productivity and quality of maintenance tasks. In our study, we observed 173 enhancement tasks done using model-based software development, and 156 enhancement tasks using traditional software development, in six live projects over one year at Robert Bosch Engineering & Business Solutions Private Ltd., a CMMi level 5 software company. For each of these tasks, we collected data on size, the effort taken to complete the task and rework effort to fix any bugs. We found that the productivity of enhancement tasks executed using model-based software development was higher by over 10% as compared to traditional software development. No statistically significant difference was found between the model-based and traditional software development for the rework effort suggesting that there is no adverse effect on quality.

6.3.1 Introduction

Reducing the gap between the requirements and the respective software implementation is important to minimize defects and improve productivity. There exist various types of models (UML diagrams [212, 213], scenario-based models [214], process-oriented models [215], etc.) to bridge this gap by capturing the necessary specifications of both requirements and implementation. However, these models fail to shield programmers from the complexities of the underlying implementation challenges of target platforms.

In model-based software development, the design gradually develops into an executable artifact with different layers of graphical abstractions called models (independent of target platform). These models can be used to automatically generate executable codes, suitable for the target platform, using code generators [216]. Examples of model-based development tools that support development of models and generating code from the models include Matlab/Simulink/State-flow [217] and Ascet [218]. In this work, we concentrate on studying the model-based development of embedded software using the tool Ascet.

Effort saved using model-based development cannot be equivalent to the manual coding effort in traditional software development, as the effort is needed for steps like developing models, integrating code generators, generating code suitable for the target platform, etc. For new software development projects, the effort spent on these steps is generally less than the total effort spent on manual coding phase (which is 40%-50% of the overall software development effort) in traditional software development, thereby providing a productivity gain with model-based software development.

Projects in software companies are majorly categorized into two types: new projects and maintenance projects. New software development projects do not have any legacy code. These projects gather all requirements, create a design, develop and test the code. However, maintenance projects are those projects where the code is maintained for fixing any bugs and adding new functionality.

Typically, product based companies will have many maintenance projects modifying the existing code for adding new requirements of customers. As mentioned earlier, model-based development may provide a productivity gain for new software development projects. However, the effect of model-based software development on the productivity of maintenance projects needs to be studied.

Enhancement tasks are those tasks that require adding new code and/or modifying the existing code of maintenance projects. Typically, a programmer has to identify first the code/models that need to be changed and then alter the code/models such that new requirements are taken care without affecting the existing functionality of old requirements. The process of executing enhancement tasks is almost same for both model-based and traditional development projects.

To understand the effect of model-based software development on the productivity of enhancement tasks, we aim to address these research questions:

RQ1: Whether model-based software development yields higher productivity than traditional software development for enhancement tasks during the maintenance phase of software projects?

RQ2: Whether any observed improvement in productivity is achieved through a compromise on the quality of the software projects?

6.3.2 Model-Based Software Development

Overall process of a model based Software development has been discussed earlier. In this process, modeling is the most important phase. We briefly describe modeling approach, before we discuss enhancements.

Various notions of modeling exist in the literature. Brown has mentioned about the spectrum of modeling: [219] the first type contain no graphical models and code is the only artifact; the second type involves generating visual models from the already existing code for understanding the code better; the third type involves developing first the models describing the system and later using those models as a reference for developing the code; the fourth type uses both code and models to update each other; the fifth type is model centric where models are the primary artifact and code is generated from those models; and finally, the sixth type involves only models and no code.

UML is considered a general-purpose modeling language that can be used across various domains. UML helps to capture both static and dynamic behaviors of the system through various diagrams. However, these diagrams are used as blue prints/sketches for developing the code (system) and are not directly used to automatically generate the code. Hence, domain-specific modeling languages with high semantics and notations (graphical) have evolved and are directly utilized to generate the code automatically from these domain-specific models [220, 221, 222]. Further, these domain-specific modeling helps to run, debug and test the models. In automotive and avionics domains, Ascet and Matlab are the two commonly used tools to develop the models.

Models are further classified into two types: platform-independent models and platform-dependent models [223]. Ascet tool allows programmers first to develop, run and test platform-independent models. Later, the models can be configured and integrated with code generators for automatically generating the code suitable for embedded controller/platform.

A “model” in model-based software development (used in embedded software) is somewhat analogous to a “function” in traditional software development. Similar to functions, a model can accept a set of inputs and process them to get required outputs, use parameters and constants, and even call other models. Further, code can be generated from each model in addition to executing/simulating each model separately.

Typically, in model-based development, each model is a block diagram with a set of inputs and outputs. A sample model containing two other models (cal_veh_speed and cal_brake_accel) is shown in Figure 6.2. The block diagram “cal_veh_speed” takes four inputs (wheel speeds of a car) and calculates the vehicle speed (output Veh_speed) which is fed to another block diagram “cal_brake_accel.” Based on the difference between the target vehicle speed (input Target_veh_speed_ip) and the present vehicle speed (veh_speed_ip), the block diagram “cal_brake_accel” sets either Brake or Accelerator.

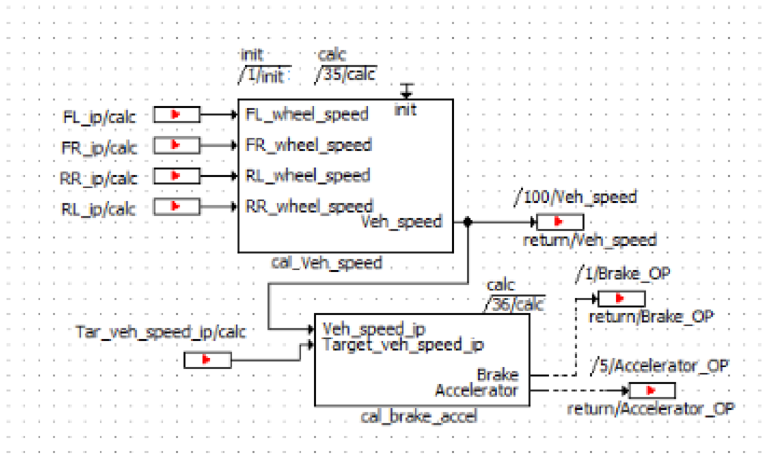


Figure 6.2: A Sample Hierarchical Model Containing Two Other Models

Software development often contains various tasks/threads running according to a required scheduling mechanism. Ascet also supports scheduling various tasks/threads according to their desired frequency and priority in both pre-emptive and non-pre-emptive modes. "calc" and "init" in Figure 6.3 are two such tasks. The models (or block diagrams) in Ascet are not executed concurrently but sequentially in discrete time. Programmers have to set the order of execution of models. For example: 35/calc and 36/calc in Figure 6.3 indicate the execution order of the models in the "calc" task/thread. Incorrect assignment of models to tasks/threads may lead to wrong results impacting quality.

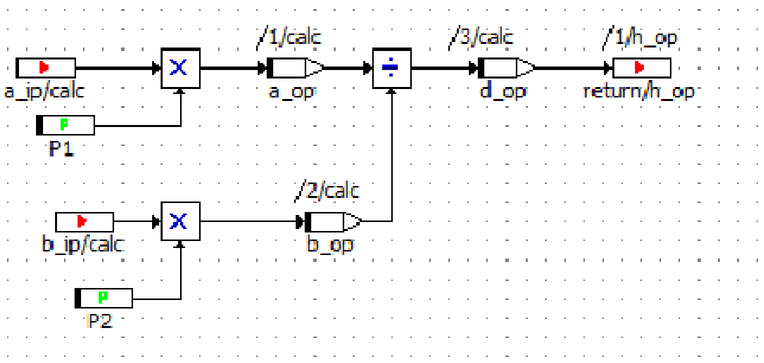


Figure 6.3: A Sample Model Capturing the Functionality

Though code is generated automatically from the models by integrating code generators, the approach is not straightforward and consumes some effort [224]. In general, any functionality is captured based on a sequence of operations/steps. Code generators are not intelligent enough to identify the correct sequence of operations defined in the models to generate the code automatically for capturing the functionality. Therefore, once the low-level design models are extended to capture the required functionality, programmers will have to number each operation in the model to inform code generator about the correct sequence of operations for generating the code [225]. Ascet has a feature to automatically assign sequential numbers for operations/steps in the model. In this case, programmers may have to verify whether the assigned sequential numbers capture the intended functionality or not.

A programmer modeling the statement h_op = (a_ip*p1)/(b_ip*p2) is shown in Figure 6.3. "a_ip," "b_ip" are inputs, "h_op" is output and "p1," "p2" are parameters/constants. The sequences 1/calc, 2/calc, 3/calc show the order of execution of operations/steps in the calc task/thread, that is, programmer first multiplied the input a_ip with parameter p1 and assigned the result to a local variable a_op. Later, the input b_ip is multiplied with parameter p2 and the result is assigned to a local variable b_op. Finally, the result of dividing a_op with b_op is assigned to another local variable d_op before assigning the same to output h_op. Wrong assignment of sequences may lead to bugs. For example, interchanging 3/calc and 2/calc lead to "divide by zero" because the variable b_op used in the calculation of d_op is zero initially and will be updated to b_ip*p2 only after the calculation of d_op.

Due to the presence of different types of embedded target controllers/platforms, several steps are required to generate code that is suitable for the target embedded controller. Some of the steps include programmers assigning correct data types to variables/parameters/constants in the model, ensuring no truncation or resolution loss occurs while performing arithmetic operations, creating and scheduling threads/processes and assigning each thread/process with functionality defined in the model.

6.3.3 Enhancements in Model-Based Software

For maintenance tasks, the overall process followed by traditional and model-based software development is similar to each other, though the methods of execution of the steps are different. An overview of the overall process followed for maintenance tasks in model-based software as well as traditional method is shown in Figure 6.4. As can be seen, the structure and logical steps are similar, but the methods of execution are different.

To add any new functionality or to modify the existing functionality during the maintenance phase, the programmer has to identify and make changes such that the required functionality is modeled appropriately without affecting the rest of the existing functionality, a process similar to that of traditional software development.

Model-based development tools provide simulation environment to compile, run and test the model being developed/changed. This step could be analogous to compiling

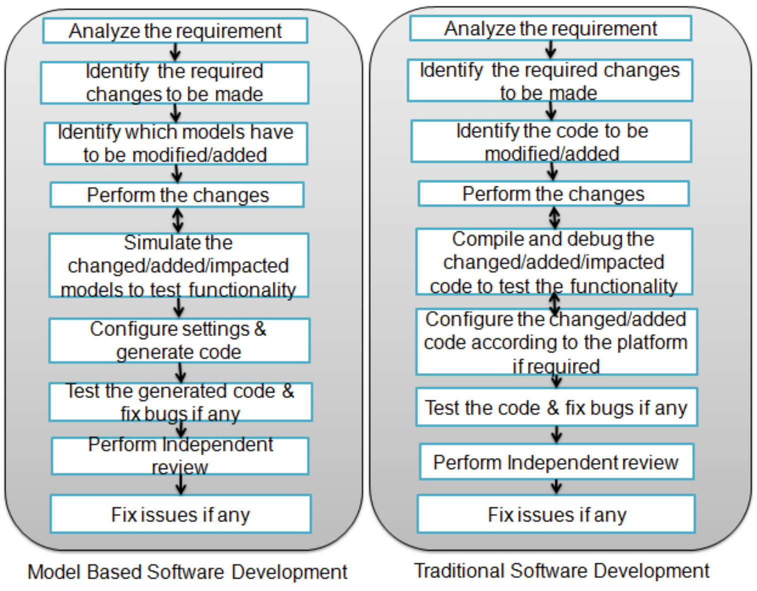

Analyze the requirement
Identify the required changes to be made
Identify which models have to be modified/added
Perform the changes
Simulate the changed/added/impacted models to test functionality
Configure settings & generate code
Test the generated code & fix bugs if any
Perform Independent review
Fix issues if any
Model Based Software Development
Analyze the requirement
Identify the required changes to be made
Identify the code to be modified/added
Perform the changes
Compile and debug the changed/added/impacted code to test the functionality
Configure the changed/added code according to the platform if required
Test the code & fix bugs if any
Perform Independent review
Fix issues if any
Traditional Software Development

Figure 6.4: Process of Enhancement Tasks in Traditional and Model-based Developments

and debugging the code in traditional software development. Though code generation settings already exist in the project (during maintenance phase), configurations like sequencing the operations, assigning models to tasks/threads have to be set for the newly added/changed models to generate code from the added/changed models.

Unlike traditional software, enhancement tasks in model-based development require testing the model as well as the auto-generated code from the model to ensure required functionality is captured competently and the auto-generated code is suitable for the target platform [172].

Testing the auto-generated code in model-based software is different from testing the manual code in traditional software development. Code generated from the model is difficult to understand, and therefore test cases have to be derived based on the models to achieve the coverage of code generated from the models. Further, no direct mapping between the generated code and the model make deriving test cases difficult. Various test design techniques exist to derive test cases. In this study, programmers used exploratory method while testing model-based and traditional software [172].

Additionally, the problems associated with model-based software development like traceability of requirements to model and generated code, understanding the highly optimized auto-generated code from model, copying differences between models, mapping defects identified in auto-generated code back to model, etc., may pose challenges on productivity of enhancement tasks in the model-based software projects compared to traditional software projects [226, 227].

6.3.4 Field Study and Data Collection

We conducted this field study at Robert Bosch Engineering & Business Solutions Ltd for about one year in India.

Selection of projects: Six similar projects that were stable and running for a long time were selected for this study. These projects belong to the same domain, follow the same organizational process, develop software at the application layer of the product and use the same configuration, project management and build tools. These projects have skilled programmers executing tasks using both model-based software development and traditional software development.

Type of tasks: Typically a maintenance project can have multiple types of tasks: bug fixings, enhancements, investigations, etc. In this study, we considered enhancement tasks.

Selection of programmers: Programmers having two to three years of experience executing enhancement tasks using model-based software development and traditional software development were selected.

Development environment: Model-based development tasks were executed using the tool Ascet. Traditional software development tasks were executed using C language.

Table 6.6: Number of tasks of different sizes in the two development models

Size (ATR)	0-5	6-10	11-15	16-20	>20
Model-based	72	53	18	12	16
Traditional	68	21	25	13	19

Independent review: Each task executed by programmers was reviewed by a senior programmer in the project.

Software size: For size, we used Adjusted Testable Requirements (ATR), a variation of the Testable Requirements metric, which is a stable and common size measure followed in the organization for enhancement tasks in both model-based and traditional software development for a very long time.

Data collection: We collected data from 329 tasks in the six chosen projects. Around 170 tasks were performed using model-based software development and 155 were performed using traditional software development. The number of tasks of different sizes in the model-based and traditional software development is given in table 6.6.

Total effort: Total effort spent on the task is the effort taken from the start to the completion of that task. A task is said to be complete when the software is archived and delivered after fixing all the non-conformance issues/defects found by an independent expert review and by thorough testing. Effort spent by each programmer was captured and compared against the working hours of the programmer spent in the organization to eliminate erroneous effort, if any, reported by the programmers.

Rework effort: Rework effort is part of the total effort and includes only the effort spent to fix the defects/nonconformance issues identified in the code/model developed. The number of defects and non-conformance issues identified in model-based projects was not compared with traditional software projects. The reason being, analysis of the defects between two different types of software development environments requires correct categorization of defects based on the type of defects, the complexity of defects, etc., hence will be subjective.

6.3.5 Analysis and Results

We compared the software sizes of tasks executed using model-based approach and tasks executed using traditional approach. After removing the outliers, Anderson--Darling test was performed to check the normality of the data. As the data did not have a normal distribution, Mann-Whitney, a nonparametric test, was performed. The results showed that the difference between the software sizes of the tasks performed using model-based and traditional software development was not statistically significant. This indicates that the bias due to the size of tasks is minimized.

Our first research question was : Does model-based development yield higher productivity than traditional software development for enhancement tasks? For this, we determined the productivity for each task using software size of the task and the total effort spent on executing the task. We found that productivity of enhancement tasks executed by model-based development (based on differences between medians) was higher by over 10% compared to the productivity of non-model based development tasks as shown in Figure 6.5.

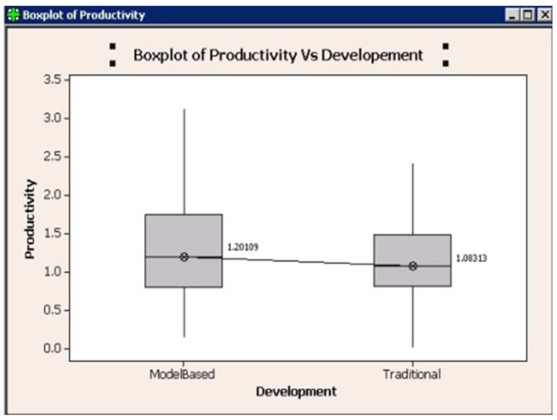


Figure 6.5: Productivity of Model-Based Development vs Traditional Software Development

We also studied the impact of the two development approaches on the productivity of tasks of different sizes. This comparison is shown in Figure 6.6. In this, the size is in ATR and the number 100 and 0 represent the percentage of generated code - so 100 represents model-based development, and 0 represents regular development. The number of tasks in each of size categories was given earlier. As we can see, the productivity of model-based development is higher in all, except in the size category 6-10 ATR, where it is about the same as for regular development. This further strengthens the analysis that productivity is better in model-based development.

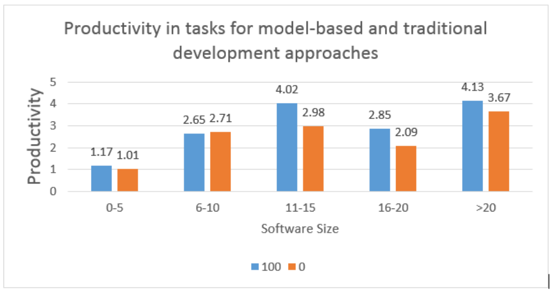


Figure 6.6: Productivity in Tasks of Different Sizes for the Two Development Approaches

Our second research question was: Is improvement in productivity achieved through compromise on quality? For this we collected effort spent by programmer to fix issues/bugs/unconformities identified in their tasks by an independent expert review. Our data revealed no significant difference between the rework efforts of model-based and traditional software development suggesting that model-based development yield high productivity without compromise on quality as shown in Figure 6.7.

To better understand the possible reasons behind this productivity improvement, we interviewed few project managers and programmers. We used semi-structured/open-ended questions to know their views on the effect of model-based development on the productivity of enhancement tasks. All the project managers and programmers whom we interviewed felt that model-based development yields higher productivity for enhancement tasks than traditional software development. This qualitatively confirmed what our study revealed.

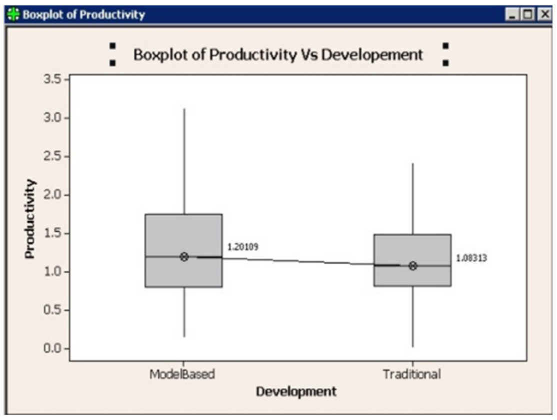


Figure 6.7: Rework Effort of Model-Based Software Development vs Traditional Software Development

Most managers and programmers felt that the main reason for productivity improvement was due to models providing graphical abstractions of the system at different hierarchical levels, helping programmers to understand and analyze the models in less time compared to traditional software development. Additionally, they felt that the simulation capability of model-based development that allows programmers to simulate each model separately at the required hierarchical level helps programmers test the functionality on-fly while incorporating the required changes in the existing models, which also helps improve productivity.

6.3.6 Summary

Model-based software development is used to a great extent in avionics and automotive domains to develop software for embedded products. The software of these embedded products is often maintained for many years; therefore, enhancing the existing functionality along with fixing bugs in the existing code base is not an easy task. To analyze the long-term sustainable benefits of model-based software development, the effect of model-based software development on the productivity of enhancement tasks need to be well understood. In this study, we observed a total of 329 enhancement tasks performed by programmers using model-based and traditional software development in six similar live projects at Robert Bosch Engineering & Business Solutions Private Ltd., a CMMi level 5 software company. We found that the productivity of model-based enhancement tasks was higher by over 10% as compared to traditional software development. Programmers and project managers whom we interviewed feel that different layers of graphical abstractions and inherent simulation capabilities of model-based software development help comprehend software better and improve productivity.

Chapter 7

Summary and Discussion

Much of the work on productivity improvement has focused on introducing a process improvement or a new tool in a project or an organization. Relatively lesser attention has been given to the behavior or practices of programmers for improving productivity.

We know that some programmers, even those with similar backgrounds, experience, and training, are much more productive as compared to others - difference can often be 2 to 3 times. In other words, the productivity of high-productivity programmers in a project sometimes can be 2 to 3 times the productivity of average-productivity programmers in the project. What makes a programmer far more productive than an average programmer is an issue that has not been well studied.

In this thesis, we focused on the impact of task processes on programmer productivity. Typically, a programmer who is assigned a task of a few days would execute it incrementally in small steps, each step performing some well-defined activity. How the execution of these steps is organized by a programmer is what we refer to as a task process. Task process used by one programmer may vary from another for the same task as the overall software process does not standardize any task process. It should, therefore, be possible to improve the productivity of programmers by improving their task processes. Further, it should be possible to improve the productivity of average-productivity programmers by training them to follow the task process of high productivity programmers. This is the focus of this thesis.

7. SUMMARY AND DISCUSSION

In this thesis, we studied the task processes of high and average-productivity programmers to investigate mainly the following research questions:

- Does a programmer use similar task process while executing a similar type of task?
- Are the task processes of high- and average-productivity programmers similar? And how do they differ from each other?
- Whether the productivity of average-productivity programmers increases by transferring the task processes of high-productivity programmers?

For studying these research questions, we studied some live projects in Robert Bosch Engineering and Business Solutions Limited, a CMMi Level 5 company. We took a few similar model-based testing projects, and in each project, we identified two sets of programmers - high-productivity programmers and average-productivity programmers. We studied the task processes of programmers in the two groups to identify the similarity between the task processes used by programmers in a group, and differences from the task processes used by the other group. Finally, we transferred the task processes of high-productivity programmers to average-productivity programmers by training the average-productivity programmers on the key steps missing in their process but commonly present in the high productivity programmers, and studied the impact of this on their productivity.

For studying the task processes of programmers, we captured their computer monitor and then analyzed the task videos to extract the task process used for a task. For quantifying the differences between task processes, we modeled each task process as a Markov chain, with each step in a task process as a node and moving from one step to another as the transition. The difference between task processes was taken to be the distance between the state transition matrices of task processes.

Our study showed that task processes of high-productivity programmers are similar to each other, while task processes of average-productivity programmers vary more.

The study also shows that task processes of high-productivity programmers are different from the task processes of average-productivity programmers.

For transferring the task processes of high-productivity programmers to average-productivity programmers, we identified the important steps that need to be changed, added, or deleted, and then trained the average-productivity programmer(s) on those. We found that the productivity of average-productivity programmers increased significantly and the similarity between the task processes of high- and average-productivity programmers also increased.

Apart from studying the task processes of high- and average-productivity programmers, we have also reported results of a few other studies for improving the productivity of programmers. We investigated the effect of productivity of trainers on the productivity of new programmers and the team by conducting a limited study on a live project. We found that the new programmers trained by high-productivity programmers were more productive than the programmers trained by average-productivity programmers. We also found a strong similarity between the task processes of the new programmers and their trainers.

We reported a study on countering Parkinson's law to improve programmer productivity. To counter the effect of Parkinson's law that may be there, we allocated one-third less time than the estimated effort for a task, while facilitating issue resolution at the same time so the main reason for wasted time is removed. We conducted this study for about six months in seven software projects, and found an improvement of at least 15% in productivity of the programmers without any degradation in the quality of the programs developed by them.

We also studied the impact of model based development on productivity and quality of maintenance tasks, in which we observed 173 enhancement tasks done using model based software development, and 156 enhancement tasks using traditional software development, in six live projects over one year. We found that the productivity of enhancement tasks executed using model based software development was higher by over 10% as compared to traditional software development.

7.1 Limitations and Threats to Validity

Our framework for studying impact of task processes requires grouping of programmers into two groups of high- and average-productive. Consequently, the framework requires a reasonable number of programmers executing a particular type of task for grouping them into two groups of high- and average-productive. Hence, this framework can be applied only to tasks like coding, enhancements, testing, etc. where more number of programmers in a project would be working. It may not be suitable for tasks in which only a few people are involved, e.g. architecture design, project planning, etc. It should however be noted that for productivity gains it is in any case more advantageous to focus on tasks where larger number of people are involved.

The framework is based on learning by average productivity programmers from their high-productivity peers. This implicitly assumes that the population on which this framework is applied is such that they are open to learning from peers and may be looking for ways to improve their task execution processes. We feel that due to this, the framework may be more suitable for junior programmers in a project rather than experienced senior programmers since the latter group is generally rather small and senior programmers are often in a state where they may have evolved and mature personal task processes which may not be amenable to change, or which may have already incorporated learnings from peers.

In our study, we applied this framework on programmers having 1-2 years of experience largely because of the direction from management as the programmers having 1-3 years of experience are the major workforce in the organization executing project tasks. (Typically, the size of a project team in the organization is around 12-14 members. The number of programmers having experience less than three years would be around 8. The remaining members would be a project manager, associate project manager, architects, and specialists, etc. It is due to this, the number of programmers in some studies is around six only from a project team.) This is a clear limitation of this work and the results - it is based only on study of junior programmers of 1-2 years of experience and it is not clear how the results may generalize for programmers with more experience. Also, the number of subjects involved in the studies is limited - that is also a threat to generalizability of the results we have seen

We have applied this framework to only model-based unit-testing tasks. It was with great deal of effort and convincing (and a study we conducted with project managers) to get a buy-in for doing this study. We tried to get the buy-in from management for applying this to other types of tasks, but that was not approved. Due to this, the results of the experiment are clearly not generalizable to other tasks executed by large number of (junior) programmers, like coding, bug fixing, enhancements, unit testing of code developed in regular programming languages, etc.

We hope that other organizations or groups will use the framework to try it on other tasks to better understand the utility of this approach. It should be added, that an attempt was made and some companies were approached for trying this, but they did not show interest in using the framework being used by one company in theirs. Also, as we have noted, even when the senior management wants to do such a study, considerable effort and thinking is needed to get the buy-in from different stakeholders involved in the experiment. We expect that if this framework is applied to other tasks executed by junior programmers, some benefits should accrue to the average-productivity programmers by learning from their higher-productivity peers.

Another key limitation of this work is that the experiments are done in one company only, and in one country. While it can possibly be argued that most high maturity organizations (in India) doing unit testing of model based development may benefit from applying this framework, the confidence in these results will increase if more experiments can be repeated in different companies, perhaps across countries also, to further validate the results.

Conducting controlled experiments on live projects in challenging. To repeat the execution of same tasks by other programmers to create a baseline for comparison is generally not possible in companies executing live projects, as it has direct impact on cost and schedule. Due to this, we did not attempt any controlled experiments, but used live projects.

The tasks in the study are similar to each other though not exactly same. Further, the tasks are distributed across the two groups almost randomly. Due to this, we feel that no systematic bias was introduced while assigning tasks to programmers. Further, unlike the tasks in normal software development, the tasks in an application layer of safety embedded software are much confined and have limited variation. For example, one task in normal software development can have pointers, and another task may not have. One task may be calling for external files or repositories etc. but another task may not be doing that. However, when it comes to an application layer of safety embedded software, the variation in tasks will be limited. The tasks in safety embedded application layer software will neither use pointers, structures not call any additional file mechanisms, etc. Additionally, the tasks considered in this study belong to the same application, same domain, same underlying platform (hardware), and use the same tools for execution.

7.2 Future Work

Much more can be done in studying task processes. First, a clear possible future work is to see the impact of task processes on other commonly assigned tasks to programmers. In particular, tasks like modeling, coding, bug fixing, enhancements, etc., in which many programmers spend a considerable amount of effort. Further, such studies, or even the repeat of the study we had done, can be done in other organizations to see if similar benefits are obtained.

In this work, we have focused on identifying the task processes of high-productivity programmers and using it to identify possible improvement opportunities in average-productivity programmers in model-based testing. We have not considered the possibilities of using understanding and analysis of task processes for improving the productivity of high-productivity programmers themselves. Once task processes are recorded, along with their attributes about how much effort is being spent on different steps in the process, it should be possible to apply process analysis and improvement approaches to explore productivity improvement for high-productivity programmers as well.

For the study of task processes, in this work, we extracted task processes through manual analysis of videos and records. It is clear that automated platforms need to be developed that can capture the video and with some inputs from the programmers, can convert it to task processes, which can then be analyzed through tools. This is another area of future work.

Another possible line of work can be to develop lightweight methods to identify task processes (eg., a programmer documents it and then somebody check if needed that is how the programmer execute the task), and identify key differences (e.g. by visually comparing and discussing), and then transfer the processes of high-productivity programmers to average-productivity programmers by having high-productivity programmers train them, particularly on the differences. Such an approach is likely to be appealing to many large organizations which have quality teams or process teams in place for improving productivity.

In this work, we focused on peer groups only. The assumption was that they have similar potential and background, so the possibility of successfully transferring task process from one sub-group to another is more feasible. However, we also know that productivity of programmer often improves with experience, at least in the initial years. There have been studies to understand how experienced programmers do some work. This framework can be used to understand their task processes as well. It is not clear how effective the transfer of the task processes will be in this case - but clearly, this is another area of conducting experiments.

A key goal of the framework for studying task processes is to improve the productivity of individual programmers. In the current work, the learning for improving the task process comes from task processes of other more productive programmers. There is a possibility of exploring other approaches for learning for improving the productivity of a programmer. For example, the learning for a programmer can be obtained from the data collected about the programmers past coding assignments through various repositories and methods. This can open a new area of “self-learning” by programmers using mining and learning for improving their productivity, or other aspects of their work.

Chapter 8

Papers Published

- Damodaram Kamma, and Pankaj Jalote. "High Productivity Programmers Use Effective Task Processes in Unit-Testing." In 2015 Asia-Pacific Software Engineering Conference, pp.32-39. IEEE, 2015.
- Damodaram Kamma and Sasi Kumar."Effect of Model Based Software Development on Productivity of Enhancement Tasks–An Industrial Study." In 2014 Asia-Pacific Software Engineering Conference, vol. 1, pp.71-77. IEEE, 2014.
- Damodaram Kamma and Pooja Maruthi, "Effective unit testing in model based software development." Proceedings of the 9th AST 2014, pp.36-42, coallocated with ICSE 2014.
- Damodaram Kamma, "Study of task processes for improving programmer productivity." DS track, pp.702-705, 2014, Hyderabad, India, ICSE 2014
- Damodaram Kamma, Geetha G and Padma Neela J, "Countering Parkinson's law for improving productivity." Proceedings of the 6th ISEC, February pp.21-23, New Delhi, India, 2013
- Damodaram Kamma and Pankaj Jalote, "Effect of task processes on programmer productivity in model-based testing." Proceedings of the 6th ISEC, pp.23-28, February 21-23, New Delhi, India, 2013.

References

[1] P. Jalote. **CMMi in Practice: Processes for Executing Software Projects at Infosys**. *Addison-Wesley Professional*, 2000. 2, 5, 6, 9, 40

[2] D. Smith W. Yu and S. Huang. **Software productivity measurement**. *AT& T Tech Journal, 69(1), pp.110-120*, 1990. 2

[3] B. Kitchenham and E. Mendes. **Productivity measurement using multiple size measures**. *IEEE Software, 30(12) , pp.1023-1035*, 2004. 2, 8

[4] IEEE Standards Board. **IEEE Standard for Software Productivity Metrics**. *IEEE Std, pp.1045-1992*, 1993. 2

[5] A. MacCormack, C. Kemerer, M. Cusumano, and B. Crandall. **Trade-Offs between Productivity and Quality in Selecting Software Development Practices**. *IEEE Software, pp. 78-79*, 2003. 3

[6] Phillip G. Armour. **Beware of counting LOC**. *Communications of the ACM, 47(3), pp.21-24*, 2004. 3

[7] Parareda B, Benedikt Mas, and Pizka M. **Measuring productivity using the infamous lines of code metric**. *Proceedings of SPACE 2007, Japan*, 2007. 3

[8] Kaushal Bhatt, Vinit Tarey, and Pushpraj Patel. **Analysis Of Source Lines Of Code(SLOC) Metric**. *Emerging Technology and Advanced Engineering 2(5), pp.150-154*, 2012. 3

[9] S. Yu and S. Zhou. **A survey on metric of software complexity**. *The 2nd IEEE International Conference on Information Management and Engineering (ICIME), pp.352-356*, 2010. 3

[10] Jorgensen M. **Experience with the accuracy of software maintenance task effort prediction models**. *IEEE Transactions of Software Engineering, 21(8), pp. 674-681*, 1995. 3

[11] Stevenson C. **Software Engineering Productivity**. *Chapman & Hall*, 1995. 3

[12] Robert Park. **Software Size Measurement: A Framework for Counting Source Statements**. *Software Engineering Institute: Technical Report SEI-92-TR-020*, 1992. 3

[13] IEEE. **Standard for Software Productivity Metrics (IEEE Std 1045)**. *The Institute of Electrical and Electronics Engineers*, 1993. 3

[14] P. Oman and J. Hagemeister. **Metrics for Assessing Software System Maintainability**. *Proc. Conf. Software Maintenance, pp. 337-344*, 1992. 3

[15] A. Abran and P. N. Robillard. **Function points analysis: An empirical study of its measurement processes**. *IEEE Transactions on Software Engineering, 22(12), pp. 895-910*, 1996. 4

[16] Sean Furey. **Why We Should Use Function Points**. *IEEE Software, 14(2), p.28*, 1997. 4

[17] D St-Pierre, M Maya, A Abran, J Desharnais, and P Bourque. **Full Function Points: Counting Practice Manual**. *Technical Report, Software Engineering Management Research Laboratory and Software Engineering Laboratory in Applied Metrics*, 1997. 4

[18] Laird L and Brennan M. **Software measurement and estimation: A practical approach**. *John Wiley & Sons, Vol. 2*, 2006. 4

[19] C. F. Kemerer. **Reliability of function points measurement: A field experiment**. *Commun. ACM, 36(2), pp. 85-97*, 1993. 4

[20] Peter B. Wilson. **Sizing software with testable requirements**. *systems development management, Auerback publications, CRC Press LLC, pp.10-34*, 2000. 4, 5, 94, 104

[21] Mosaic. **Testable requirements**. *http://www.mosaicinc.com/mosaicinc/html/tr_sizer.htm*. 5

[22] M. Arnold and P Pedross. **Software size measurement and productivity rating in a large scale software development department**. *International IEEE conferenece on software engineering, Los Alamitos, CA, USA, pp.490-493*, 1998. 5

[23] S. Nageswaran. **Test Effort Estimation Using Use Case Points**. *14th International Software Internet Quality Week, Available from http://www.cognizant.com/cogcommunity/presentations/Test_Effort_Estimation*, 2001. 5

[24] R. Clemmons. **Project estimation with use case points**. *The Journal of Defense Software Engineering, 19(2) , pp. 18-22*, 2006. 5

[25] M. Cohn. **Estimating with use case points**. *Methods & Tools, 13(3), pp. 3-13*, 2006. 5

[26] S. Moser and O. Nierstrasz. **The Effect of Object-Oriented Frameworks on Developer Productivity**. *Computer, 29(9), pp. 45-51*, 1996. 5

[27] K.D. Maxwell. **Applied Statistics for Software Managers**. *Software Quality Institute Series, Prentice-Hall*, 2002. 5

[28] E. R. C. de Almeida, B. T. de Abreu, and R. Moraes. **An Alternative Approach to Test Effort Estimation Based on Use Cases**. *In Proceedings of International Conference on Software Testing, Verification, and Validation, pp.279-288*, 2009. 5

[29] Capers Jones. **By popular demand: Software estimating rules of thumb**. *Computer, 29(3), pp.116-118*, 1996. 5

[30] E. Stensrud and I. Myrtveit. **Identifying High Performance ERP Projects**. *IEEE Transactions on Software Engineering, 29(5), pp. 398-416*, 2003. 5, 6

[31] Barbara Kitchenham and Emilia Mendes. **Software Productivity Measurement**. *IEEE Transactions on Software Engineering, 30(12), pp.1023-1035*, 2004. 5, 6

[32] K. Molokken and M. Jorgensen. **A review of surveys on software effort estimation**. *International Symposium on empirical software engineering, pp.223-230*, 2003. 5

[33] Barry Boehm, Chris Abts, and Sunita Chulani. **Software development cost estimation approaches- A survey**. *Annals of software engineering, 10(1), pp. 177-205*, 2009. 5

[34] Damodaram Kamma, Geetha G, and Padma Neela J. **Countering Parkinson's law for improving productivity**. *Proceedings of the 6th ISEC, pp. 91-96*, 1996. 6, 33

[35] M. Jorgensen. **Practical guidelines for expert-judgement-based software effort estimation**. *IEEE Software, 22(3), pp. 2-8*, 2005. 6

[36] Martin Shepperd, Chris Schofield, and Barbara Kitechenham. **Effort estimation using analogy**. *Proceedings of the 18th international conference on Software engineering, pp. 170-178*, 1996. 6

[37] B. W. Boehm. **Software engineering economics**. *Englewood Cliffs, NJ: Prentice-Hal*, 1981. 6

[38] G. E. Wittig and G. R. Finnie. **Using Artificial Neural Networks and Function Points to Estimate Software Development Effort**. *Australian Journal of Information Systems, 1(2), pp. 87-94*, 1994. 6

[39] Irina Diana Coman, Alberto Sillitti, and Giancarlo Succi. **A case-study on using an Automated In-process Software Engineering Measurement and Analysis system in an industrial environment**. *31st International Conference on Software Engineering, pp.81-99*, 1995. 6, 7

[40] IBM. **Timesheet Software**. *http://www.basic.co.ukl/timesheets.htm*. 6

[41] Andre N. Meyer, Laura E. Barton, Gail C. Murphy, Thomas Zimmermann, and Thomas Fritz. **The Work Life of Developers: Activities, Switches and Perceived Productivity**. *IEEE Transactions on Software Engineering*, 2017. 6, 19

[42] P.M. Johnson and A.M. Disney. **A critical analysis of PSP data quality: Results from a case study**. *Journal of Empirical Software Engineering, 4(4), pp.317-349*, 1999. 6, 26

[43] Sapience. **Automatic effort capturing tool**. *http://sapience.net*. 6, 7

[44] RescueTime. **Time Tracking Software**. *http://www.rescuetime.com/*. 6

[45] Emily I. M. Collins, Jon Bird, Anna L. Cox, and Daniel Harrison. **Social networking use and RescueTime: the issue of engagement**. *In Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing Adjunct Publication , pp.687-690*, 2014. 7

[46] P. M. Johnson, H. Kou, J. M. Agustin, C. Chan, C. A. Moore, J. Miglani, S. Zhen, and W. E. Doane. **Beyond the personal software process: Metrics collection and analysis for the differently disciplined**. *International Conference on Software Engineering, Portland, Oregon, pp.641-646*, 2003. 7, 26

[47] GrindStone. **Time Tracking Software**. *http://www.epiforge.com/grindstone/*. 7, 48

[48] G. Mark, D. Gudith, and U. Klocke. **The cost of interrupted work: more speed and stress**. *in Proceedings of the SIGCHI conference on Human Factors in Computing Systems, pp. 107-110*, 2008. 7

[49] B. P. Bailey, J. A. Konstan, and J. V. Carlis. **The effects of interruptions on task performance, annoyance, and anxiety in the user interface**. *in Proceedings of INTERACT, vol. 1, pp. 593-601*, 2011. 7, 8, 20

[50] T. DeMarco and T. Lister. **Peopleware. Productive Projects and Teams**. *Dorset House Publishing, New York*, 1987. 7

[51] W. D. Brooks. **Software technology payoff: Some statistical evidence**. *J. Syst. Software, 2(1), pp.3-9*, 1981. 7

[52] R. H. Rasch. **An investigation of factors that impact behavioural outcomes of software engineers**. *Proceedings of SIGCPR, ACM, pp. 38-53*, 1991. 7

[53] J. D. Blackburn, G. D. Scudder, and L. N. Van Wassenhove. **Improving speed and productivity of software development: A global survey of software developers**. *IEEE transactions on software engineering, 22(12), pp.875-885*, 1996. 7

[54] L. Macaulay. **The importance of human factors in planning the requirements capture stage of project**. *International Journal of Project Management, 15(1), pp.39-53*, 1997. 7

[55] Port D and McArthur M. **A study of productivity and efficiency for object-oriented methods and languages**. *Asia Pacific Software Engineering Conference, pp. 128-135*, 1999. 7

[56] R. Berntsson-Svensson and A. Aurum. **Successful software project and products: An empirical investigation**. *ISESE, ACM Press, pp.144-153*, 2006. 8

[57] K. Spiegl. **Projektmanagement Life - Best Practices und Significant Events im Software-Projektmanagement**. *Universitat Wien*, 2007. 8

[58] M. K. Goncalves, L. R. de Souza, and V. M. Gonzalez. **Collaboration, information seeking and communication: An observational study of software developers work practices**. *Jounal of UCS, 17(14), pp.1913-1930*, 2011. 8, 20

[59] Ronald H Rasch and Henry L Tosi. **Factors Affecting Software Developers' Performance: An Integrated Approach**. *Management Information Systems Research Center, 16(3), pp. 395-413*, 1992. 8

[60] W. Hayes and J. W. Over. **The Personal Software Process (PSP): An empirical study of the impact of PSP on individual engineers**. *Technical Report CMU/SEI-97-TR-001, Software Engineering Institute, Pittsburgh, PA*, 1997. 8, 25

[61] W.S. Humphrey. **Using a defined and measured personal software process**. *IEEE Software, pp.77-88*, 1996. 8, 25, 26

[62] T. McGibbon. **A business case for software process improvement revised**. *DACS State-of-the-Art Report, Rome Laboratory*, 1999. 8

[63] Marcelo Cataldo and James D Herbsleb. **Coordination Breakdowns and Their Impact on Development Productivity and Software Failures**. *IEEE Software, 39(3), pp.343-360*, 2012. 8

[64] T Bruckhaus, N.H. Madhavii, I Janssen, and J Henshaw. **The impact of tools on software productivity**. *IEEE Software, 13(5), pp.29-38*, 1996. 8

[65] Thomas Tan, Qi Li, Barry Boehm, Ye Yang, Mei He, and Ramin Moazeni. **Productivity trends in incremental and iterative software development**. *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 1-10*, 2009. 8

[66] S. Wagner and M. Ruhe. **A systematic review of productivity factors in software development**. *language*, 1989. 8

[67] Premraj R, Shepperd M, Kitchenham B, and Forselius P. **An empirical analysis of software productivity over time**. *11th IEEE International symposium on software metrics*, 2005. 9

[68] Kulpa MK and Johnson KA. **Interpreting the CMMI (R): A Process Improvement Approach**. *CRC Press*, 2008. 9

[69] Team CP. **Capability Maturity Model Integration**. *(CMMI), Version 1.1,Continuous Representation*, 2002. 9

[70] Team CP. **Capability Maturity Model Integration**. *(CMMI), Version 1.2,Continuous Representation*, 2006. 9

[71] BK Clark. **Quantifying the effects of process improvement on the effort**. *IEEE Software, 17(1), pp.65-70*, 2000. 9

[72] M. S. Krishnan D. E. Harter and S.A. Slaughter. **Effects of process maturity on quality, cycle time and effort in software product development**. *Management Science, 46 (1), pp.451-466*, 2000. 9

[73] D. Galin and M. Avrahami. **Are CMM program investments beneficial? Analyzing past studies**. *IEEE Software, 23(6), pp.81-87*, 2006. 9

[74] M. Diaz and J. Sligo. **How software process improvement helped Motorola**. *IEEE Software,14(5), pp.75-81*, 1997. 9

[75] Agrawal M and Chari K. **Software effort, quality, and cycle time: A study of CMMi level 5 projects**. *IEEE Transactions on software engineering, 33(3)*, 2007. 9

[76] Murphy C.N. **The International Organization for Standardization (ISO): global governance through voluntary consensus**. *Routledge*, 2009. 9

[77] M. Paulk and ldquo. **Comparing ISO 9001 and the capability maturity model for software**. *Software quality journal, pp.245-256*, 1993. 10

[78] Jovanovic V and D Shoemaker. **ISO 9001 standard and software quality improvement**. *Benchmarking for Quality Management & Technology, 4(2), pp.148-159*, 1997. 10

[79] Chattopadhyay Satya P. **Improving the speed of ISO 14000 implementation: a framework for increasing productivity**. *Managerial Auditing Journal, 16(1), pp.36-40*, 2001. 10

[80] Mueller M, Hoermann K, Dittmann L, and Zimmer J. **Automotive SPICE in Practice: surviving implementation and assessment**. *O'Reilly Media, Inc.*, 2008. 10

[81] T. K. Varkoi and T. K. Mkinen. **Case study of CMM and SPICE comparison in software process assessment**. *IEMC Proceedings, Pioneering New Technologies: Management Issues and Challenges in the Third Millennium, Puerto Rico, USA, pp.477-482*, 1998. 10

[82] Vanamali Bhaskar, Fabio Bella, and Klaus Hormann. **From CMMI to SPICE-Experiences on How to Survive a SPICE Assessment Having Already Implemented CMMI**. *Computer Software and Applications, pp.1045-1052*, 2008. 10

[83] Randimbivololona F. **Orientations in verification engineering of avionics software**. *In Informatics Springer, Berlin Heidelberg, pp. 131-137*, 2001. 10

[84] V. Hilderman and T. Baghai. **Avionics Certification - A complete guide to DO-178 (Software) DO-254 (Hardware)**. *Avionics Communications Inc., VA, USA*, 2007. 10

[85] Basili V R. **The maturing of the Quality Improvement Paradigm**. *in the SEL*, 1993. 10

[86] G. Caldiera V. Basili and H.D. Rombach. **Goal Question Metric Approach**. *Encyclopedia of Software Engineering, John Wiley & Sons, New York, pp.528-532*, 1994. 10, 12

[87] Hendricks C A and Kelbaugh R L. **Implementing six sigma at GE**. *The Journal for Quality and Participation, 21(4), p.48*, 1998. 10, 13

[88] Kan Stephen H. **Metrics and models in software quality engineering**. *Addison-Wesley Longman Publishing Co., Inc.*, 2002. 11

REFERENCES

[89] Shull Forrest J, Jeffrey C Carver, Sira Vegas, and Natalia Juristo. **The role of replications in empirical software engineering**. *Empirical Software Engineering, 13(2), pp.211-218*, 2008. 11

[90] Tonini, Antonio Carlos, Mauro De Mesquita Spinola, and Fernando Jose Barbin Laurindo. **Six Sigma and software development process: DMAIC improvements**. *In Technology Management for the Global Future, Vol.6, pp.2815-2823*, 2006. 13

[91] Deming W Edwards. **Out of the crisis, Massachusetts Institute of Technology**. *Center for advanced engineering study, Cambridge, MA*, 1986. 13

[92] P. S. Pande, R. P. Neuman, and R. R. Cavangh. **The Six Sigma Way: How GE, Motorola, and Other Top Companies Are Honing Their Performance**. *McGraw Hill, New York*, 2000. 13

[93] R.G. Schroeder. **Six Sigma quality improvement: what is Six Sigma and what are the important implications?** *Proceeding of the Fourth Annual International POMS Conference, Seville, Vol.27, Spain*, 2000. 13

[94] Cronemyr Peter. **DMAIC and DMADV-differences, similarities and synergies**. *International Journal of Six Sigma and Competitive Advantage, 3(3), pp.193-209*, 1986. 13

[95] Andries De Grip and Jan Sauermann. **Scientific management**. *Routledge*, 2004. 14

[96] Andries De Grip and Jan Sauermann. **Taylorism transformed: Scientific management theory since 1945**. *UNC Press Book*, 2016. 14

[97] Andries De Grip and Jan Sauermann. **Time-and-motion regained**. *Harvard Business Review, 71(1), pp.97-108*, 1993. 14

[98] Liker J.K. and Morgan J.M. **The Toyota way in services: the case of lean product development**. *The Academy of Management Perspectives, 20(2), pp.5-20*, 1993. 14

[99] D. M Boje and R. D. Winsor. **The resurrection of Taylorism: Total quality managements hidden agenda**. *Journal of Organizational Change Management, 6(4), pp.57-70*, 2000. 15

[100] I. Kato and A. Smalley. **Toyota Kaizen Methods: Six Steps to Improvement**. *CRC Press Taylor & Fracis*, 2011. 15

[101] Abdulmalek F.A. and Rajgopal J. **Analyzing the benefits of lean manufacturing and value stream mapping via simulation: A process sector case study**. *International Journal of production economics, 107(1), pp.223-236*, 2007. 15

[102] Rahani A.R. and Al-Ashraf M. **Production flow analysis through value stream mapping: a lean manufacturing process case study**. *Procedia Engineering, 41, pp.1727-1734*, 2007. 16

[103] S.H Kan. **Metrics and models in software quality engineering**. *Addison-Wesley Longman Publishing Co*, 2007. 16

[104] Hartley Stephen. **Project Management: principles, processes and practice**. *Pearson Education*, 2009. 16

[105] W.W. Agresti. **Software engineering as industrial engineering**. *Software Eng. Notes, 6(5), pp.11-12*, 1981. 16

[106] W.S. Humphrey. **The personal software process: Status and trends**. *IEEE Software,17(6), p.72*, 2000. 17

[107] Acemoglu D, Gancia G, and Zilibotti F. **Competing engines of growth: Innovation and standardization**. *Journal of Economic Theory, 147(2), pp.570-601*, 2012. 17

[108] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. **How Effective Developers Investigate Source Code: An Exploratory Study**. *IEEE Transactions on Software Engineering, 30(12)*, 2004. 18

[109] ROBERT W. BOWDIDGE and WILLIAM G. GRISWOLD. **How Software Engineering Tools Organize Programmer Behavior During the Task of Data**. *Empirical Software Engineering, pp. 221-267*, 1997. 18

[110] Joseph Lawrance, Christopher Bogart, Margaret Burnett ad Rachel Bellamy, Kyle Rector, and Scott D. Fleming. **How Programmers Debug, Revisited: An Information Foraging Theory Perspective**. *IEEE Transactions on Software Engineering, 39(2), pp.889-903*, 2013. 19

[111] Roman Bednarik and Markku Tukiainen. **An eye-tracking methodology for characterizing program comprehension processes**. *Proceedings of the 2006 symposium on Eye tracking research and applications, pp. 125-132*, 2006. 19

[112] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. **An examination of software engineering work practices**. *CASCON' 10. IBM Corporation,2010, pp.174-188*, 2010. 19

[113] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann. **Software developers perceptions of productivity**. *in Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014. ACM, 2014, pp.19-29*, 2014. 19

[114] R Van Solingen, E. Berghout, and F. Van Latum. **Interrupts: just a minute never is**. *IEEE software,15(5), pp.97-103*, 1998. 19

[115] M. Czerwinski, E. Horvitz, and S. Wilhite. **A diary study of task switching and interruptions**. *in Proceedings of the SIGCHI conference on Human factors in computing systems, pp.175-182*, 2004. 20

[116] Andrew. J. Ko, R. DeLine, and G. Venolia. **Information needs in collocated software development teams**. *in Proceedings of the 29th International Conference on Software Engineering, pp. 344-353*, 2007. 20

[117] R. Minelli, A. Mocci, and M. Lanza. **I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time**. *23rd IEEE International Conference on Program Comprehension, pp. 25-35*, 2015. 20

[118] H. Sanchez, R. Robbes, and V. M. Gonzalez. **An empirical study of work fragmentation in software evolution tasks**. *in 2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering, pp. 251-260*, 2015. 20

[119] E. E. Grant and H.Sackman. **An exploratory investigation of programmer performance under on-line and off-line conditions**. *IEEE transactions on Human Factors in Electronics, 8(1), pp.33-48*, 1967. 20

[120] L. Prechelt. **An empirical study of working speed differences between software engineers for various kinds of task**. *IEEE Transactions on Software Engineering, 33(10), pp.23-29*, 2000. 21

[121] G. K. Gill and C. F. Kemerer. **Productivity Impacts of Software Complexity and Developers Experience**. *IEEE Transactions on Software Engineering*, 1990. 21

[122] G. K. Gill and C. F. Kemerer. **Quantitative and qualitative differences between experts and novices in chunking computer software knowledge**. *Int. J. Hum.-Comput. Interact, 6(1), pp.105-118*, 1994. 21

[123] B. P. Bailey, J. A. Konstan, and J. V. Carlis. **Conceptual data modelling in database design: similarities and differences between expert and novice designers**. *International Journal of Man-Machine Studies, 37(1), pp. 83-101*, 1992. 21

[124] B. P. Bailey, J. A. Konstan, and J. V. Carlis. **Differences between Novice and Expert Systems Analysts: What Do We Know and What Do We Do?** *International Journal of Management Information Systems, 15(1), pp. 9-50*, 1998. 21

[125] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. **What would other programmers do: suggesting solutions to error messages**. *SIGCHI Conference on Human Factors in Computing Systems, Atlanta, Georgia, USA, pp.1019-1028*, 2010. 21, 22

[126] L. Gugerty and G. Olson. **Debugging by skilled and novice programmers**. *CHI'86 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 171-174*, 1986. 21

[127] L. Gugerty and G. Olson. **Novice/expert differences in programming skills**. *International Journal of Man-Machine Studies, 23(4), pp. 383-390*, 1985. 21

[128] M.E. Crosby, J. Scholtz, and S Wiedenbeck. **The roles beacons play in comprehension for novice and expert programmers**. *In 14th Workshop of the Psychology of Programming Interest Group, pp. 58-73*, 2002. 22

[129] K.Torii, K. I Matsumoto, K. Nakakoji, Y. Takada, S. Takada, and K. Shima. **Ginger2: An environment for computer-aided empirical software engineering**. *IEEE Transactions on Software Engineering, 25(4), pp.472-492*, 1999. 22

[130] M. Kersten and G. C. Murphy. **Using task context to improve programmer productivity**. *in Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 1-11*, 2014. 22

[131] Y.H. Kim, J. H. Jeon, E. K. Choe, B. Lee, K. Kim, and J. Seo. **TimeAware: Leveraging Framing Effects to Enhance Personal Productivity**. *in Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems , pp. 272-283*, 2016. 23

[132] R. C. Martin and R. S. Koss. **An extreme programming episode: Advanced Principles, Patterns and Process of Software Development**. *Prentice Hal*, 2001. 23

[133] Marat Boshernitsan, Susan L. Graham, and Marti A. Hearst. **Aligning Development Tools with the Way Programmers Think About Code Changes**. *CHI 2007 Proceedings, pp.567-576*, 2007. 23

[134] J. Kamatar and W. Hayes. **An experience report on the personal software process**. *IEEE Software, 17(6), pp.85-89*, 2000. 25

[135] P.M. Johnson and A.M. Disney. **The personal software process: A cautionary case study**. *IEEE Software, 15(6), pp.85-88*, 1998. 26

[136] K. E. Emam, B. Shostak, and N. Madhavji. **Implementing concepts from the personal software process in an industrial setting**. *Fourth international conference on software process, pp.117-130*, 1996. 26

[137] W.S. Humphrey. **Applying the PSP in industry**. *IEEE Software, 17(6), pp.90-95*, 2000. 26

[138] W.S. Humphrey. **How to study individual programmers**. *22nd International Conference on Software Engineering*, 2000. 26

[139] G. Canfora, A. Cimitile, and C. Aaron. **Lessons learned about distributed pair programming: What are the knowledge needs to address**. *In proceedings of 12th IEEE international workshop on enabling technologies: infrastructure for collaborative enterprises, pp. 314-319*, 2003. 26

[140] S. Pietinen, V. Tenhunen, and M. Tukiainen. **Productivity of pair programming in a distributed environment - results from two controlled case studies**. *European Conference on Software Process Improvement, pp. 47-58*, 2008. 26

[141] A. Parrish, R. Smith, D. Hale, and J. Hale. **A field study of developer pairs: Productivity impacts and implications**. *IEEE Software, 21(5), pp. 76-79*, 2004. 27

[142] Kim Man Lui and Keith Chan. **Pair programming productivity: Novice-novice vs. expert-expert**. *International Journal of Human-Computer Studies, pp.915-925*, 2006. 27

[143] F. Padberg and M.M. Muller. **Analyzing the cost and benefit of pair programming**. *In proceedings of ninth international software metrics symposium, pp. 166-177*, 2003. 27

[144] Cockburn A and Williams L. **The costs and benefits of pair programming**. *Extreme programming examined, pp. 223-247*, 2000. 27

[145] Dyba T, Arisholm E, Sjoberg, Hannay, and Shull F. **Are two heads better than one? On the effectiveness of pair programming**. *IEEE Software, 24(6), pp.12-15*, 2007. 27

[146] B. Al-Ani, D. Redmiles, A. van der Hoek, M. Alvim, I. Almeida da Silva, N. Mangano, E. Trainer, and A. Sarma. **Continuous Coordination within Software Engineering Teams: Concepts and Tool Support**. *Journal of Computer Science and Engineering in Arabic,1(3), pp.10-33*, 2008. 27

[147] N Mangano and A. Van der Hoek. **The design and evaluation of a tool to support software designers at the whiteboard**. *Automated Software Engineering, 19(4), pp. 381-421*, 2012. 27

[148] N Mangano, T.D. LaToza, M Petre, and A Van der Hoek. **How Software Designers Interact with Sketches at the Whiteboard**. *IEEE Transactions on Software Engineering, 41(2), pp.135-156*, 2015. 27

[149] H. Erdogmus, M. Morisio, and M. Torchiano. **On the effectiveness of the test-first approach to programming**. *IEEE Transactions on software Engineering,31(3), pp. 226-237*, 2005. 27

[150] Maximilien E.M. and Williams L. **Assessing test-driven development at IBM**. *Proceedings. 25th International Conference In Software Engineering, pp.564-569*, 2003. 27

[151] Madeyski, Lech, and Lukasz Szaa. **The impact of test-driven development on software development productivity - an empirical study**. *In European Conference on Software Process Improvement, pp.200-211*, 2007. 27

[152] Ahmed A, Ahmad S, Ehsan N, Mirza E, and Sarwar S Z. **Agile software development: Impact on productivity and quality**. *IEEE International Conference in Management of Innovation and Technology, pp.287-291*, 2010. 27

[153] Iivari J. **Why are CASE tools not used?** *Communications of the ACM, 39(10), pp.94-103*, 1996. 27

[154] Lim Wayne C. **Effects of reuse on quality, productivity, and economics**. *IEEE software, 11(5), pp.23-30*, 1994. 27

[155] Basili Victor R, Lionel C Briand, and Walcelio L Melo. **How reuse influences productivity in object-oriented systems**. *Communications of the ACM, 39(10), pp.104-116*, 1994. 27

[156] Tripathi, Akash Kumar, and Atul Gupta. **A controlled experiment to evaluate the effectiveness and the efficiency of four static program analysis tools for Java programs**. *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, pp.23*, 2014. 28

[157] Saurabh Tiwari and Atul Gupta. **A Controlled Experiment to Assess the Effectiveness of Eight Use Case Templates**. *20th Aisa Pacific Software Engineering Conference, Vol.1, pp.207-214*, 2013. 28

[158] Atul Gupta and Pankaj Jalote. **An Approach for Experimentally Evaluating Effectiveness and Efficiency of Coverage Criteria for Software Testing**. *International Journal of Software Tools and Technology Transfer,10(2), pp.145-160*, 2008. 28

[159] Cain J W and McCrindle R J. **An investigation into the effects of code coupling on team dynamics and productivity**. *In Computer Software and Applications Conference, pp.907-913*, 2002. 28

[160] Bisant David B and James R Lyle. **A two-person inspection method to improve programming productivity**. *IEEE Transactions on Software Engineering, 15(10), p.1294*, 2002. 28

[161] Yasutaka Kamei, Audris Mockus, and et.al. **large-scale empirical study of just-in-time quality assurance**. *IEEE Transaction on software engineering , 39(6), pp. 757-773*, 2012. 28

[162] Ray, Baishakhi, Meiyappan Nagappan, Christian Bird, Nachiappan Nagappan, and Thomas Zimmermann. **The uniqueness of changes: characteristics and applications**. *In Proceedings of the 12th Working Conference on Mining Software Repositories, pp. 34-44*, 2015. 28

[163] P.J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and B Murphy. **Not my bug! and other reasons for software bug report reassignments**. *In Proceedings of the ACM 2011 conference on Computer supported cooperative work, pp.395-404*, 2011. 28

[164] F. Thung, D. Lo, L. Jiang, F. Rahman, and P Devanbu. **When would this bug get reported?** *In 28th IEEE International Conference on Software Maintenance (ICSM), pp.420-429*, 2012. 28

[165] Kazuhiro yamshita, Audris Mockus, and et.al. **Thresholds for Size and Complexity Metrics: A Case Study from the Perspective of Defect Density**. *IEEE Software, 33(4), pp. 40-45*, 2016. 28

[166] Abram Hindle, Christian Bird, Thomas Zimmermann, and Nachiappan Nagappan. **Do topics make sense to managers and developers?** *In Journal of Empirical Software Engineering, 20(2), pp.479-514*, 2015. 28

[167] Jialiang Xie, Audris Mockus, and Minghui zhou. **Impact of triage: A study of Mozilla and Gnome**. *Emperical Software Engineering and Measurement, pp.247-250*, 2013. 28

[168] A. Pasala, S. Guha, G. Agnihotram, and S Padmanabhuni. **An Analytics-Driven Approach to Identify Duplicate Bug Records in Large Data Repositories**. *In Data Science and Big Data Computing, pp.161-187*, 2016. 28

[169] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V Filkov. **Quality and productivity outcomes relating to continuous integration in GitHub**. *In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 805-816*, 2015. 28

[170] Vasilescu B, Blincoe K, Xuan Q, Casalnuovo C, Damian D, Devanbu P, and Filkov V. **The sky is not the limit: multitasking across GitHub projects**. *In Proceedings of the 38th International Conference on Software Engineering, pp.994-1005*, 2015. 28

[171] Damodaram Kamma. **Study of task processes for improving programmer productivity**. *In Companion Proceedings of the 36th International Conference on Software Engineering , pp.702-705*, 2014. 30

[172] K. Damodaram and M. Pooja. **Effective unit-testingin model-based software development**. *Proceedings of the 9th AST, pp.36-42*, 2014. 31, 138

[173] K. Damodaram and J. Pankaj. **Effect of task processes on programmer productivity in model-based testing**. *India Software Engineering Conference, pp.23-28*, 2013. 31

[174] K. Damodaram and J. Pankaj. **High productivity programmers use effective task processes in unit-testing**. *Asia-Pacific Software Engineering Conference, pp.32-39*, 2015. 31

[175] K. Damodaram and G. Sasikumar. **Effect of model based software development on productivity of enhancement tasks - an industrial study**. *21st Asia-Pacific Software Engineering Conference, pp.71-77*, 2014. 33

[176] Sira Vegas, Oscar, and Natalia Juristo. **Difficulties in Running Experiments in the Software Industry: Experiences from the Trenches**. *IEEE/ACM 3rd International Workshop on Conducting Empirical Studies in Industry, pp. 3-9*, 2015. 36

[177] Dag I K Sjoberg, Jo E Hannay, Ove Hansen, Vigdis Kampenes, Amela Karahansanovic, Nils Kristian, and Annette C Rekdal. **A survey of controlled experiements in software engineering**. *IEEE Transactions on Software Engineering, 31(9), pp.733-753*, 2005. 36

[178] P. Jalote and G. Jain. **Assigning tasks in a 24-h software development model**. *Journal of Systems and Software, 79(7), pp. 904-911*, 2006. 40

[179] Hemant Kr. Meena, Indradeep Saha, Koushik Kr. Mondal, and T. V. Prabhakar. **An Approach to Workflow Modeling and Analysis**. *OOPSLA, pp. 85-89*, 2005. 42

[180] Amit Raj, Ashish Agrawal, and T. V. Prabhakar. **Transformation of business processes into UML models: an SBVR approach**. *International Journal of Scientific & Engineering Research, 4(3), pp.647-661*, 2013. 42

[181] B. Curtis, M. I. Kellner, and J. Over. **Process modeling**. *Communications of the ACM, 35(9), pp.75-90*, 1992. 42

[182] Osterweil Leon. **Software processes are software too**. *9th International Conference on Software Engineering, Monterey, California, USA, pp.2-13*, 1987. 42

[183] M. Suzuki and T. Katayama. **Metaoperations in the process model HFSP for the dynamics and flexibility of software processes**. *First International Conference on the Software Process. IEEE Computer Society, Washington, DC, pp.202-207*, 1991. 42

[184] M. Suzuki and T. Katayama. **Software process modeling: A behavioural approach**. *10th International Conference on Software Engineering, pp.174-186*, 1988. 42

[185] M. Suzuki and T. Katayama. **A behavioural approach to software process modelling**. *4th International Software Process Workshop on Representing and Enacting the Software Process, Devon, United Kingdom, pp.167-170*, 1988. 42

[186] K.E. Huff and V.R. Lessor. **Software process model evolution in the SPADE Environment**. *IEEE Transactions on Software Engineering, 19(12), pp.1128-1144*, 1993. 42

[187] K.E. Huff and V.R. Lessor. **A planbased intelligent assistant that supports the software development process**. *Third Software Engineering Symposium on Practical Software Development Environments. Software Eng. Notes, 13(5), pp.97-106*, 1989. 42

[188] Snag-It. **https://www.techsmith.com/snagit.html**. *home page.* 48

[189] M. P. E. Heimdahl, M. W. Whalen, and A. Rajan. **On mc/dc and implementation structure: An empirical study**. *Proc. of the 27th Digital Avionics Systems Conference, pp. 5-B*, 2008. 60

[190] J. Guan, J. Outt, and P. Ammannl. **An industrial case study of structural testing applied to safety critical embedded software**. *2006 ACM/IEEE International Symposium on Empirical Software Engineering, pp.272-277*, 2006. 60

[191] M. Whalen A. Rajan and M. Heimdahl. **The effect of program and model structure on mc/dc test adequacy**. *30th International Conference on Software engineering, pp.161-170*, 2008. 60

[192] K. Kapoor and J. P. Bowen. **Experimental evaluation of the variation in effectiveness for dc, fpc and mc/dc test criteria**. *ACM-IEEE International Symposium on Empirical Software Engineering, Rome, Italy, pp.185-194*, 2003. 60

[193] A. Dupuy and N.Leveson. **An empirical evaluation of the mc/dc coverage criterion on the hete-2 satellite software**. *Digital Aviation Systems Conference, Vol. 1, pp.1B6-1*, 2000. 60

[194] J. A. Jones. **Test-suite reduction and prioritization for modified condition/decision coverage**. *IEEE International Conference on Software Engineering, 29(3), pp.195-209*, 2001. 60

[195] Aldrich William. **Using model coverage analysis to improve the controls development process**. *AIAA Modeling and Simulation Technologies Conference and exhibit*, 2002. 60

[196] J. Chilenski. **An investigation of three forms of the modified condition decision coverage (mcdc) criterion**. *Technical Report DOT/FAA/AR-01/18, Ofice of Aviation Research, Washington, D.C*, 2001. 60

[197] J. Chilenski and S. P. Miller. **Applicability of modified condition/decision coverage to software testing**. *Software Engineering Journal, 9(1), 191-200*, 1994. 60

[198] M. Conrad, I. Fey, and S. Sadeghipour. **Systematic model-based testing of embedded automotive software**. *Electronic Notes in Theoretical Computer Science, Vol. 111, pp.13-26*, 2006. 63

[199] B. Sericola. **Markov Chains: Theory and Applications**. *John Wiley & Sons*, 2013. 86

[200] R. Argote, P. Ingram, J. M.Levine, and R.L. Moreland. **Knowledge transfer in organizations: Learning from the experience of others**. *Organizational behavior and human decision processes, 82(1), pp.1-8*, 2000. 116

[201] R. Argote and P. Ingram. **Knowledge transfer: A basis for competitive advantage in firms**. *Organizational behavior and human decision processes, 82(1), pp.150-169*, 2000. 116

[202] J. Pankaj. **Software project management in practice**. *Editorial Addison Wesley*, 2002. 117

[203] Vasudeva Varma, Kirti Garg, and STP Vamsi. **A Case Study Initiative for Software Engineering Education**. *International Conference on Software Engineering Research and Practice*, 2005. 118

[204] Vasudeva Varma and Kirti Garg. **Case Studies: The Potential Teaching Instruments for Software Engineering**. *International Conference on Quality Software, pp.279-284*, 2005. 118

[205] Kirti Garg and Vasudeva Varma. **A Study of the Effectiveness of Case Study approach in Software Engineering Education**. *Conference on Software Engineering Education and Training, pp.309-316*, 2007. 118

[206] Amir ELnaga and Amen Imran. **The effect of training on employees performance**. *European Journal of Business and Management, 5(4), pp.137-147*, 2013. 118

[207] Afshan Sultana, Sobia Irum, Kamran Ahmed, and Nasir Mehmood. **Impact of training on employee performance: a study of telecommunication sector**. *interdisciplinary journal of contemporary research in business, 4(6), pp.646-661*, 2012. 118

[208] Ann P. Bartel. **Training, Wage Growth and Job Performance: Evidence from a company database**. *Journal of Labor Economics, 13(3), pp. 401-425*, 1995. 118

[209] Andries De Grip and Jan Sauermann. **The Effects of Training on Own and Co-worker Productivity: Evidence from a Field Experiment**. *The Economic Journal, 122(560), pp.376-399*, 2012. 118

[210] E. A. Smith. **The role of tacit and explicit knowledge in the workplace**. *Journal of knowledge Management, 5(4), pp.311-321*, 2001. 118

[211] J. Howells. **Tacit knowledge**. *Technology analysis & strategic management, 8(2), pp.91-106*, 1996. 118

[212] A. Nugroho and M. R. V. Chaudron. **The impact of uml modeling on defect density and defect resolution time in a proprietary system**. *Empirical Software Engineering, 19(4), pp.926-954*, 2014. 132

[213] J. Rumbaugh, I. Jacobson, and G. Booch. **The Unified Modeling Language**. *Addison-Wesley*, 1998. 132

[214] A. G. Sutcliffe and N. A. M. Maiden. **Supporting scenario-based requirements engineering**. *IEEE Software, 24(12), pp.1072-1088*, 1998. 132

[215] J. Mylopoulos, L. Chung, and B. Nixon. **Representing and using non-functional requirements: A process - oriented approach**. *IEEE Software, 18(6), pp.483-497*, 1992. 132

[216] G. Karsai, A. Ledeczi J. Sztipanovits, and T. Bapty. **Model-integrated development of embedded software**. *IEEE, 91(1) ,pp.145-164*, 2003. 132

[217] Mathworks. **http://www.mathworks.com/products/simulink**. *home page*. 133

[218] Ascet. **http://www.etas.com/de/products/ascet_software_products.php**. *home page*. 133

[219] A.W.Braun, J.Conallen, and D Tropeano. **modeling and model driven architecture (mda): Model-Driven Software Development**. *Springer, Heidelberg, pp.1-16*, 2005. 134

[220] A. Van Deursen, P. Klint, and J. Visser. **Domain-specific languages: An annotated bibliography**. *ACM SIGPLAN Notices, 35(6), pp.26-36*, 2000. 134

[221] M. Mernik, J. Heering, and A. Sloane. **When and how to develop domain-specific languages**. *ACM Computing Surveys, 37(4), pp.316-344*, 2005. 134

[222] S. Kelly and J. Tolvanen. **Domain-Specific Modeling. Enabling Full Code Generation**. *John Wiley & Sons, New York*, 2008. 134

[223] S. Kelly and J. Tolvanen. **Model driven engineering**. *International Conference on Integrated Formal Methods, pp.286-298*, 2002. 135

[224] A. F. Ferrari and S. Gnesi. **Lessons learnt from the adoption of formal model-based development**. *NASA Formal Methods Symposium, pp. 24-38*, 2012. 136

[225] R. G. G. Cattell. **Formalization and automatic derivation of code generators**. *Department of Computer Science. Carnegie-Mellon, Tech. Rep. CMU-CS-78-11S*, 1978. 136

[226] R. G. G. Cattell. **The pragmatics of model-driven development**. *IEEE Software, Vol. 20, pp.19-24*, 2003. 139

[227] R. France and B. Rumpe. **Model-driven development of complex software: A research roadmap**. *Future of software engineering, pp.37-54*, 2007. 139