UNIVERSITY OF THE PHILIPPINES
OPEN UNIVERSITY

DOCTOR OF COMMUNICATION

BEN HUR C. VIRAY

A GROUNDED THEORY OF DEVELOPER COMMUNICATION IN EXTREME PROGRAMMING PRACTICES

Thesis Adviser:

MELINDA DP. BANDALARIA, Ph.D.
Faculty of Information and Communication Studies

31 May 2024

Permission of the classification of this academic work access is subject to the provisions of applicable laws, the provisions of the UP IPR policy and any contractual obligations:

Invention (I)	☐	Yes	or	■ No
Publication (P)	☐	Yes	or	■ No
Confidential (C)	☐	Yes	or	■ No
Free (F)	■	Yes	or	☐ No

Student's signature:

Dissertation adviser' signature:

University Permission Page

A GROUNDED THEORY OF DEVELOPER COMMUNICATION IN EXTREME PROGRAMMING PRACTICES

"I hereby grant the University of the Philippines a non-exclusive, worldwide, royalty-free license to reproduce, publish and publicly distribute copies of this Academic Work in whatever form subject to the provisions of applicable laws, the provisions of the UP IPR policy and any contractual obligations, as well as more specific permission marking on the Title Page."

"I specifically allow the University to:

a. *Upload a copy of the work in the theses database of the college/school/institute/department and in any other databases available on the public internet*
b. *Publish the work in the college/school/institute/department journal, both in print and electronic or digital format and online; and*
c. *Give open access to the work, thus allowing "fair use" of the work in accordance with the provision of the Intellectual Property Code of the Philippines (Republic Act No. 8293), especially for teaching, scholarly and research purposes.*

Ben Hur C. Viray
31 May 2024

Acceptance Page:

This paper prepared by **BEN HUR C. VIRAY** with the title: **"A GROUNDED THEORY OF DEVELOPER COMMUNICATION IN EXTREME PROGRAMMING PRACTICES"** is hereby accepted by the Faculty of Information and Communication Studies, U.P. Open University, in partial fulfillment of the requirements for the degree Course.

_____ _____

MELINDA DP. BANDALARIA, Ph.D.
Chair, Dissertation Committee (Date)

_____ _____

ALEXANDER G. FLOR, Ph.D.
Member, Dissertation Committee (Date)

_____ _____

RIA MAE H. BORROMEO, Ph.D.
Member, Dissertation Committee (Date)

DIEGO S. MARANAN, Ph.D.
Dean
Faculty of Information and Communication Studies

_____
(Date)

Biographical Sketch

When Ben Hur got his first computer in high school, his passion for programming ignited and has never waned. He has primarily worked as a developer, leveraging his bachelor's degree in computer science from UP Diliman. His academic background led him to complete a Master of Science in Computer Science from the same university. He also shared his knowledge by teaching Computer Science and Multimedia Studies courses at UP Diliman and UP Open University for nearly five years. With over 20 years of experience in the IT industry, he works as an SAP ABAP Specialist in a renewable energy company.

Acknowledgement

The researcher would like to express his sincerest gratitude to the following who helped throughout the several stages (and years) of his dissertation:

His adviser, Dr. Melinda DP. Bandalaria, deserves special mention for her significant contributions. Despite her busy schedule as UPOU Chancellor, she provided patient guidance and invaluable feedback, particularly on Grounded Theory, which greatly enriched this research.

The members of his panel, Dr. Alexander G. Flor and Dr. Ria Mae H. Borromeo, deserve recognition for their constructive comments and insights. Their thoughtful contributions significantly enhanced the quality of this research, making it a more comprehensive and robust study.

The IT professionals who generously shared their experiences through the questionnaire, especially those interviewed, played crucial roles in this research. Their insights significantly contributed to the findings, particularly in XP practices.

The FICS Dean and DComm program chair, Dr. Diego S. Maranan and Dr. Jean A. Saludadez, respectively, for approving his request for a waiver of the MRR so he can complete his overdue dissertation.

His DComm professors and core classmates, including the late Dr. Felix R. Librero, helped him navigate the complex world of communication.

The past and present DComm and FICS staff, whose dedicated support was instrumental throughout this study.

His immediate family, mainly his mother, Buena C. Viray, gave inspiration with their unwavering belief in him and her nurturing of a love for learning.

Finally, HIM for all His blessings and for making all these possible.

Dedication

The researcher dedicates this dissertation to his loving wife, Anne Aizza A. Viray, whose steadfast encouragement, support, and understanding were a constant source of strength throughout his journey. She always believed in him even when he doubted himself.

In Memoriam

Ruben O. Viray (1944-2024) was the researcher's courageous father. He fought until the end and inspired the researcher to persevere. He attended all his son's graduations; he would have been surprised and ecstatic with his doctorate.

TABLE OF CONTENTS

List of Figures

List of Tables

Abstract

There is evidence that Extreme Programming (XP) software development practices are still actively used by Information Technology (IT) projects under the Agile methodology, one of the famous software development life cycle (SDLC) models. However, more information about the communication between programmers or developers for these XP practices is needed. Thus, this study, employing a constructivist grounded theory (CGT) design which emphasizes the construction of knowledge through the interpretation of data, was conducted to explore developer communication for significant XP practices.

For the initial data, the researcher was fortunate to have the participation of eleven out of fifteen IT professionals, each with at least a decade of work experience. Their expertise was invaluable in shaping the direction of the research. Three out of four identified developers also participated in a follow-up questionnaire. Subsequently, semi-structured key informant interviews were conducted, focusing on their top three practices: Planning Game, Test-Driven Development (TDD), and Small Releases, which were selected based on their perceived importance in IT projects.

During the interviews, it became evident that the developers play multiple roles, each significantly influencing communication. These fourteen identified roles highlight their work's diverse and complex nature: Assessor, Collaborator, Coordinator, Designer, Developer, Documenter, Interpreter, Learner, Mentor, Negotiator, Presenter, Researcher, Tester, and Translator. The developers interacted with seven teams categorized into four groups based on their modes of communication: Direct Managers and Change management under Linear-Interactional, Business Team under Interactional, External Developers, Functional

Resources, and Technical Team under Interactional-Transactional, and Transactional-Interactional for Internal Developers.

With all the combined practices, roles, and interactions, this study has proposed an XP Developer Communication (XPDC) model. This framework can serve as a guide for understanding and improving developer communication in XP projects. For instance, it can help identify the key roles and their communication needs or guide the selection of appropriate communication modes for different teams. Although not all XP practices are covered, the components are enough to cover all the essential developer communication practices in XP.

Keywords: Developer communication; Extreme programming; Agile methodology; Software Development Life Cycle

Chapter I

INTRODUCTION

Background of the Study

SDLC is a framework that guides organizations involved in software projects. The stages involved in SDLC vary depending on the nature of the project, but the basic breakdown is planning, analysis, design, and implementation. There are also various SDLC models, the common ones being waterfall, rapid application development (RAD), and Agile development.

Unlike the other models, Agile uses an adaptive approach dependent on open communication. There are several approaches to Agile, but the popular ones are Scrum, Kanban, and XP. The focus is on XP as it involves the best software development practices, which include writing good code through Code Refactoring, Pair Programming, and TDD. Aside from these, other essential practices such as Planning Game and Small Releases are based on two of XP's core values: Communication and Feedback.

Many organizational factors influence company projects, but effective communication is necessary for success. Usually, large projects have multiple stakeholders that communicate within and across corporate teams, such as analysts, designers, developers, testers, and project managers. Different types of communication occur between and across levels, from project planning to customer support, including external entities such as consumers and third-party vendors.

Ineffective communication can lead to disastrous scenarios such as rework, delays, poor software quality, client dissatisfaction, and project failure. Various studies have analyzed the IT project communication processes to prevent these

casualties. Most researchers emphasize interaction between different IT groups; this study has decided to focus on developers or programmers who write computer code to develop software applications.

As software development is a collaborative process, developer communication is essential. Aside from programming, communicating is a required skill of a developer. In small projects, developers usually work alone, thus having no communication barriers. However, in huge ones, there are many instances where communication between developers happens, such as continuous integration, project handovers, remote meetings, and team huddles.

Although many SDLC models exist, Agile methods differ from the rest as they are more people oriented. Even though Agile focuses on communication, developers still face communication challenges. Various papers discuss effective communication aimed at different levels of Agile (Yermolaieva, 2020; Pinho & Aguiar, 2020) and within different stages in an IT professional's career: from a newly employed IT employee up to a CIO (Kappelman, Jones, Johnson, McLean & Boonme, 2016).

Communication research in XP is not usually developer-centric; some have the customers in the center, such as the role of customers in XP (Martin, 2009) and the impact of customer communication on XP defects (Korkola, Abrahamsson & Kyllonen, 2006).

Others have the global software development (GSD) teams in mind, where one study examined the communication practices of the team but did not focus on the individual practices of XP and how they were used (Layman, Williams, Damian, & Bures, 2006). Another investigated the benefits and challenges of adopting XP for GSD through a systematic literature review and surveys (Shah & Amin, 2013). Perhaps the closest research material found is how XP practices benefit

communications in a GSD team with onshore customers and an offshore development team (Xiaohu, Bin, Zhijun, & Maddineni, 2004).

For communication among developers, some papers have discussed helpful techniques in the workplace (Zieris & Prechelt, 2020; D'Angelo & Begel, 2017; Urai, Umezawa, & Osawa, 2015) and the university (Zarb, Hughes, & Richards, 2015) via pair programming, which is just one of the XP practices. There is also a mixed-method dissertation (Kumar, 2016) that has qualitatively analyzed the communication practices and formal processes of software development communities in Agile and not necessarily XP. What is lacking is research on the communication aspect of developers in relevant XP practices, a theory designed to look at XP from the programmers' communication perspective.

Research Problems

According to Charmaz (2006), this grounded theory question from Glaser should guide the study, "What is happening here?". Therefore, the research problems are:

1. Which XP practices involve the developers the most?
2. What are the different ways developers communicate in XP practices?
3. How do developers communicate in XP practices?
4. What model can be constructed for developer communication in XP practices?

Objectives of the Study

The following research objectives address the questions mentioned in the previous section:

1. Define the XP practices that involve the developers.
2. Identify the ways developers communicate in XP practices.
3. Describe how developers communicate in XP practices.
4. Develop a model for developer communication in XP practices.

Significance of the Study

Developing a grounded theory of XPDC could strengthen existing XP practices and contribute to the development of programmers. Organizations can use this research as a supplement to the usage of XP that can motivate them to invest in the communication skills of programmers, as effective communication not only prevents misunderstandings and minimizes rework but can also boost their productivity, build their morale, and make them efficient which translate into cost savings. As the researcher advocates education, this work also aims to introduce students and new developers to some best practices in programming to prepare them for the communication challenges of the software industry.

The author hopes to contribute to the field of communication by introducing a new strand that focuses on a specific demographic: the developers. The model of communication that will emerge from the study can be a contributing factor to both technology-mediated communication and organizational communication.

Scope and Limitations

Most studies focus on professional developers, but this research also involves traditional or pragmatic programmers and non-traditional end-user programmers. The former do not completely follow all XP practices and are not affiliated with any Agile method (Thomas & Hunt, 2019), while the latter develop programs with little or no programming language knowledge (Srinivasan, Jetcheva, & Chander, 2017).

As this involves qualitative methods, this research offers limited generalization of findings due to the small number of participants who have their own biases and prejudices. The interviews were conducted with individuals known to the researcher, and their experiences may not be extendable to other developers. Further research is required to sufficiently say that the theory can be applied to a larger population.

Chapter II

REVIEW OF RELATED LITERATURE

Computer-Mediated Communication (CMC) Theories

CMC has been adapted from mediated-communication theories that describe traditional media such as print, radio, and television and then applied to new media such as computers.

Uses and Gratifications Theory (UGT). UGT was first introduced by Blumler & Katz (1974) in the early 1940s. It seeks to comprehend why individuals opt for specific mass media, their underlying needs, and the satisfaction they derive from these media. In the contemporary context, Papacharissi and Rubin (2000) expanded UGT to elucidate the motivations behind Internet usage, identifying five key motives: interaction, leisure, information acquisition, convenience, and entertainment.

Social Presence Theory (SPT). Created by Short, Williams, and Christie (1976), SPT helps us understand the perception of people interact with as "real." Social presence is our ability to interpret nonverbal cues from interactions (Wood & Smith, 2005). In CMC, various forms of media will elicit various degrees of social presence from people who use different platforms. Due to technological advances, some people experience high levels of social presence in online environments such as virtual worlds (Wrench & Punyanunt-Carter, 2007).

Media Richness Theory (MRT). As Daft and Lengel (1983) proposed, richness is "the potential information carrying capacity of data". They proposed MRT as a framework to describe the levels of information richness of different media used in communication. MRT can rank certain communication technologies based on their

richness or ability to transmit needed information, including nonverbal behaviors and social cues.

Social Information Processing Theory (SIP). Salancik and Pfeffer (1978) introduced SIP to explain how social information affects job attitudes and motivation in a social context. The decision-making process is based on social factors such as past behavior and what others think about them. Walther (1992) has applied SIP in the context of CMC to understand the development of interpersonal relationships online. He argues that these CMC relationships can develop like traditional face-to-face (F2F) relationships over time, although it may take longer to achieve.

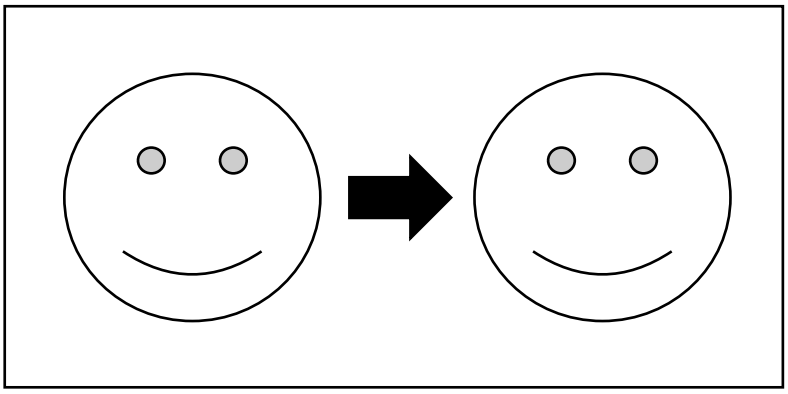
Models of Communication

Communication is an essential aspect of every organization working towards common goals. Therefore, the flow of information that contributes to the functioning of an organization merits its field of study. The field of organizational communication has vastly expanded in scope and depth through the years. In this study, we will cover only the three main communication models and some prominent models under them.

Linear Model. Shannon and Weaver (1949) developed the linear model through a "mathematical model" of communication. Information travels in a straight line using its five essential elements: source, transmitter, channel, receiver, and destination. Another linear transmission model came out a few years later (Berlo, 1960) called the SMCR model. It derived its name from the four primary communication components: source, message, channel, and receiver. Each component has several key attributes that "humanize" the model and differentiate it from Shannon and Weaver's.

This diagram can represent the linear models:

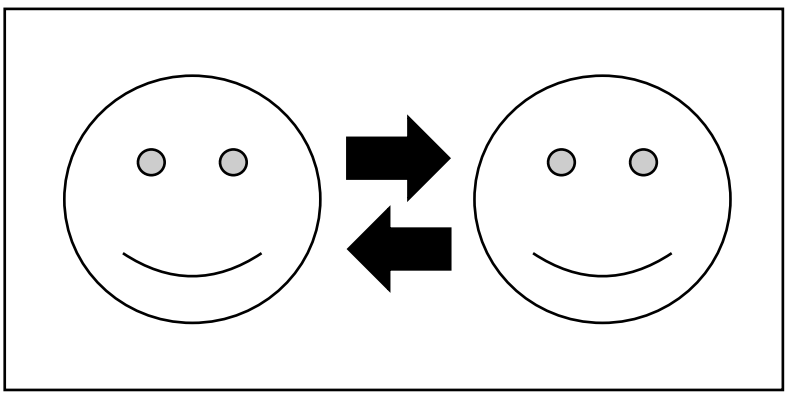
Figure 2-1: *Linear Model*



Interactional Model. In between the period of Shannon-Weaver's and Berlo's models, there was an alternative model proposed by Schramm (1954) that portrayed communication as interactional. The model has three essential components: source, destination, and message, where the latter can come out as feedback. As both participants can be senders and receivers of messages/feedback, communication has changed from linear to circular, in which information goes back and forth.

Here is an illustration of the interactional model:

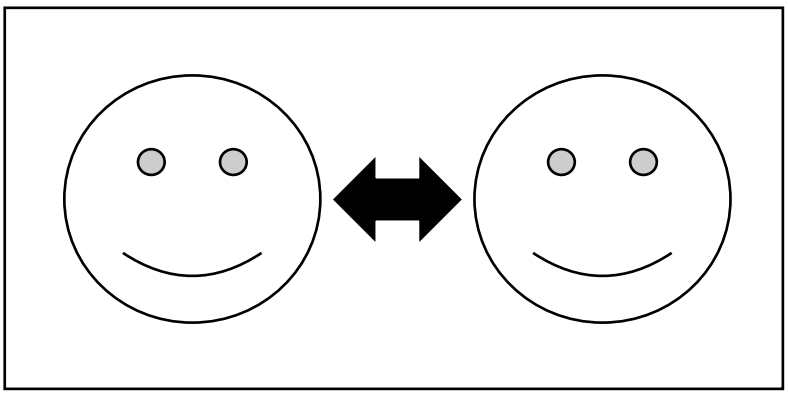
Figure 2-2: *Interactional Model*



Transactional Model. A decade after Berlo's model came out, Barnlund (1970) published a transactional communication model to address the earlier models' limitations. Unlike in Schramm's, where the participants take turns communicating, in Barnlund's, the messages/feedback are simultaneously exchanged between the conversation members. This model also relies on nonverbal cues, so it works best with communication technologies with a high degree of media richness (Drew, 2023).

The transactional model can be portrayed in the figure below:

Figure 2-3: *Transactional Model*



Software Development Life Cycle

SDLC has been introduced in computer science classes and seems simple in theory, but it is not. For an easy programming task, it is straightforward. However, building an information system is quite complex – users' business requirements are constantly changing, so the design and function can move through its phases in different ways: consecutively, incrementally, iteratively, or in other patterns. Still, they go through the same four main stages mentioned earlier.

The planning phase is the first step in understanding why and how to build the project. The analysis phase deals with most questions: who will use the project, what

it will do, and when and where it will be used. In the design phase, the stakeholders decide how the project will run. Finally, there is an implementation, where the project is constructed or customized.

The models or methodologies needed to implement the SDLC were mentioned earlier. Many organizations have their own internal methodologies, while some are formal and/or proprietary standards used by government agencies and consulting firms. While there are many models available, only the predominant ones that have evolved throughout time will be described (Dennis, Wixom, & Roth, 2012).

Waterfall. The waterfall methodology or the linear model has sequential phases (Figure 2-4). There are two variants of this methodology: parallel development and V-model. The former divides some phases into subphases that can be implemented in parallel (Figure 2-5). The latter focuses on different levels of testing, where each level is linked to the analysis or design phases (Figure 2-6).

Figure 2-4: *Linear*

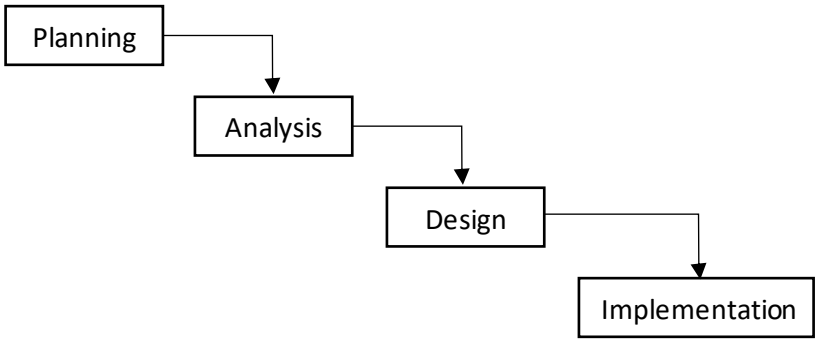


Figure 2-5: *Parallel*

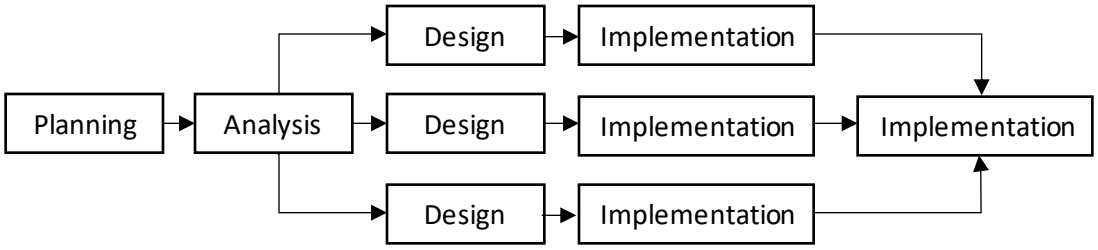
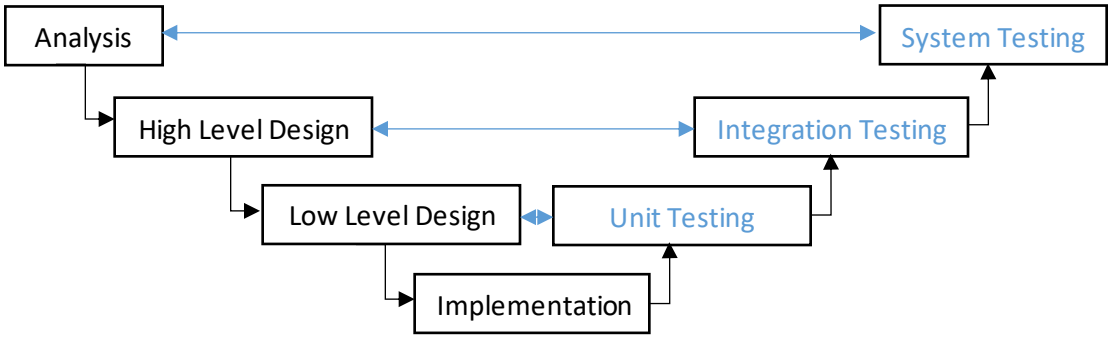


Figure 2-6: *V-Model*



Rapid Application Development. RAD responded to the weak waterfall model and its variations. It involves special tools and techniques to enhance and speed up the analysis, design, and implementation phases. There are two common RAD variants: iterative and prototyping. The first one cuts the project into a series of sequentially developed versions (Figure 2-7). The second applies the analysis, design, and implementation phases concurrently to give a beta version of the system for quick user evaluation and feedback (Figure 2-8).

Figure 2-7: *Iterative Variant*

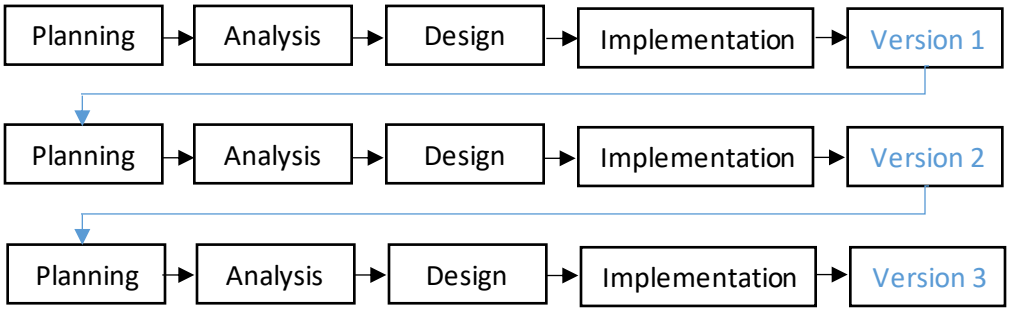
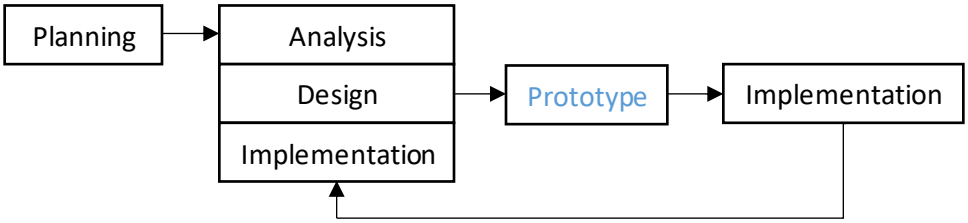
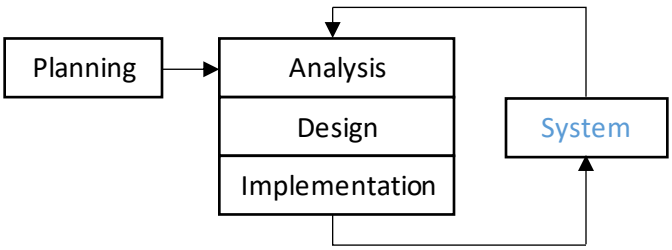


Figure 2-8: *Prototyping Variant*



Agile. Agile Development is a group of methodologies focusing on communication to eliminate modeling and documentation overhead. Agile is not a set of tools or a single methodology, but a philosophy put to paper (Agile Manifesto) in 2001 by the Agile Alliance composed of 17 people (Beck, Beedle, van Bennekum, Cockburn, Cunningham, Fowler, Grenning, Highsmith, Hunt, Jeffries, Kern, Marrick, Martin, Mellor, Schwaber, Sutherland, & Thomas, 2001). It focuses on simple, iterative developments where each iteration or cycle is considered a complete project (Figure 2-9). As the project team adapts to the current business processes, cycles are kept short. Several Agile frameworks exist, including Scrum, Kanban, Lean, Bimodal, Hybrid, Crystal, and XP.

Figure 2-9: *Agile Framework*



Agile Methodology

Agile has a manifesto that has values and principles that provide the foundations of various frameworks, including XP (Beck et al., 2001).

Values. According to the manifesto, they are "*uncovering better ways of developing software by doing it and helping others do it*." They value:

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

While the items on the right are valued, the items on the left (in bold) are valued much more.

Principles. The manifesto also has these principles (some are reworded for clarity):

- *Our paramount focus is customer satisfaction, achieved through the early and continuous delivery of valuable software.*
- *Agile processes are designed to embrace changing requirements, even in the later stages of development, leveraging them for the customer's competitive advantage.*

- *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for a shorter timescale.*
- *Business people and developers must work together daily throughout the project.*
- *Build projects around motivated individuals. Give them the environment and support they need and trust them to do the job.*
- *F2F conversation is the most efficient and effective method of conveying information to and within a development team.*
- *Working software, in the context of Agile, refers to functional software that meets the user's needs and is ready for use. It is the primary measure of progress in Agile development.*
- *Agile processes empower sponsors, developers, and users, enabling them to maintain a constant pace indefinitely, a testament to their crucial role in sustainable development.*
- *Continuous attention to technical excellence and good design is not just a principle but a driving force that inspires and motivates us to enhance our agility.*
- *Simplicity--the art of maximizing the amount of work not done--is essential.*
- *The best architectures, requirements, and designs emerge from self-organizing teams.*

These values and principles guide the Agile frameworks, including XP.

Modes of Communication. One of the initiators of Agile Software Development, Alistair Cockburn (2002), described in his book the various modes of communication based on MRT (Figure 2-10). Scott Ambler (2002) modified this to compare the effectiveness of various modes of communication with the richness of the communication channel (Figure 2-11). Ambler and Vizdos (2008) also administered a survey that explored the effectiveness of communication strategies between Agile software developers in a team and between team members and stakeholders (Table 2-1). The results showed that F2F communication is the most effective, but it is almost tied to F2F on the whiteboard for team communication. Ambler (2014) researched further by developing Figure 2-12 to choose the best communication strategy for any situation. Based on this, F2F with a sketching environment is the most effective communication method.

Figure 2-10: *Effectiveness of different modes of communication (Cockburn, 2002)*

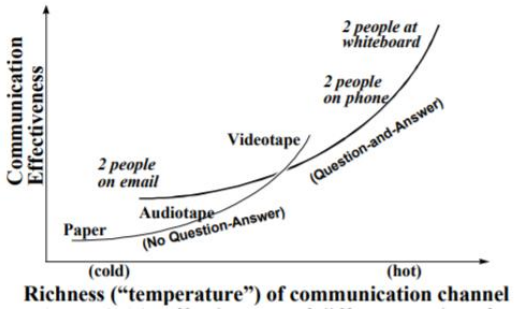


Figure 2-11: *Modes of Communication (Ambler, 2002)*

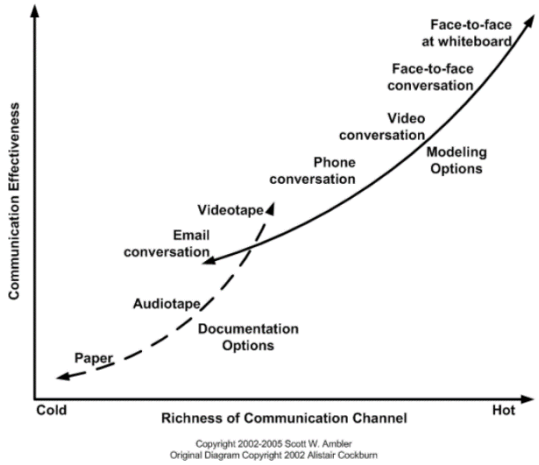
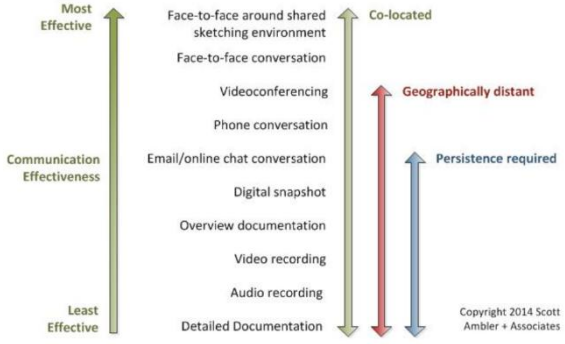


Table 2-1: *Effectiveness of Communication Strategies (Ambler, 2014)*

	Within Team	With Stakeholders
Face to face (F2F)	4.25	4.06
F2F at Whiteboard	4.24	3.46
Detailed Documentation	-0.34	0.16
Email	1.08	1.32
Overview documentation	1.84	1.86
Overview diagrams	2.54	1.89
Online chat	2.10	0.15
Teleconference calls	1.42	1.51
Videoconferencing	1.34	1.62

Copyright 2008 Scott W. Ambler www.ambysoft.com/surveys/

Figure 2-12: *Implication of the modes of communication (Ambler & Vizdos, 2008)*



Extreme Programming

Kent Beck, one of the founders of Agile, developed XP while working on the Comprehensive Compensation System payroll project at Chrysler during the 1990s (Copeland, 2001). Among the Agile frameworks, XP focuses explicitly on the developers as it encompasses the best methodologies in software engineering to empower them with tools to adapt to changing customer requirements even at the last minute. XP aims to produce high-quality software using practices taken to "extreme" levels, hence the name. XP is distinguished from other Agile methodologies by its reliance on collaboration and communication, including tests and source code, to communicate system structure and intent (Beck & Andres, 2004).

XP is a discipline of software development based on values, principles, and practices (Beck, 1999; Beck & Andres, 2004). In October 1999, Beck published a book on XP with four values, five fundamental principles, ten less central principles, and 11 practices. Five years later (2004), he and Andres published a second edition with five main values, 14 principles, 13 primary practices, and 11 corollary practices. Let us call them old and new versions to distinguish between the two.

Values. These values serve as a compass, guiding a person's behavior based on knowledge and understanding. They are practical guides that steer individuals toward achieving their software development goals.

The new version has five main values, where the first four came from the old version:

- Communication
- Simplicity

- Feedback
- Courage
- Respect

Other minor values such as safety, security, predictability, and quality of life are added (Beck & Andres, 2004). The three relevant values for this research are Communication, Simplicity, and Feedback.

- Communication - Sometimes, problems and errors are caused by a lack of knowledge, which can usually be solved through communication. XP aims to keep information flowing by having practices that involve communication. The most effective communication is F2F, but updated code and documents can also be reliable.
- Simplicity - When faced with ever-changing requirements, sometimes it is better to keep it simple. It is better to do simple things that will be used for now rather than have an unusable complex solution. Simplicity complements communication because the simpler the system, the less a person has to communicate. On the other hand, a person achieves simplicity by focusing only on the needed requirements by improving communication.
- Feedback - In software development, even a well-planned project can have changes; these trigger feedback. Feedback is a critical communication component, as the communication cycle is incomplete without it. Feedback can come in different forms, such as the system's state, code completion, or testing progress. It also complements simplicity since the simpler the solution, the easier it is to get feedback.

Principles. The values are somehow vague in guiding our behaviors, so the principles come in here. They are intellectual techniques for translating values into practice, bridging the gap between the two (Beck, 1999).

In the old version, the five fundamental principles include:

- Rapid feedback
- Assume simplicity
- Incremental change
- Embracing change
- Quality work

The less central principles are:

- Teach learning
- Small initial investment
- Play to win
- Concrete experiments
- Open, honest communication
- Work with people's instincts, not against them
- Accepted responsibility
- Local adaptation
- Travel light
- Honest measurement

In the new version, Beck and Andres (2004) reworded almost all the principles except for the 12th and last entries on this list:

- Humanity

- Economics
- Mutual Benefit
- Self-Similarity
- Improvement
- Diversity
- Reflection
- Flow
- Opportunity
- Redundancy
- Failure
- Quality
- Baby Steps
- Accepted Responsibility

The relevant principles from the old version are rapid feedback and open and honest communication. These are mutual benefit, improvement, diversity, and flow for the new. Accepted responsibility is also essential in both the old and new versions. As two of the old relevant principles are similar to the previous values, the focus is on the new principles (Beck & Andres, 2004).

- Mutual Benefit - Perhaps the most important XP principle is that the activities should benefit all concerned now and in the future. For example, proper technical documentation that includes unit testing will benefit not only the change owners but also future programmers who will maintain or change the code.

- Improvement - Software development could be better, so continuous improvement is essential. In the quest for perfection, activities must be refined, and this is achievable with the feedback process in place.
- Diversity - When working on a project, many teams with various skills and personalities are involved. With diversity comes conflict, which does not always have a negative connotation because it can involve clashing of ideas, like when proposing solutions. Conflicts, whether good or bad, can be resolved through communication.
- Flow - Being flow-oriented means having a continuous flow of activities to achieve steady development until release. Instead of delivering in big chunks, which is risky, the principle of flow suggests deploying smaller increments instead. Any deviations or disruptions to the flow should be resolved, and this is where feedback comes into play to make sure the changes are going in the right direction.
- Accepted Responsibility - Responsibility can only be accepted and not assigned. Depending on the role, the individual should be responsible enough to decide if a given task is acceptable. Any misalignments can distort the team's communication.

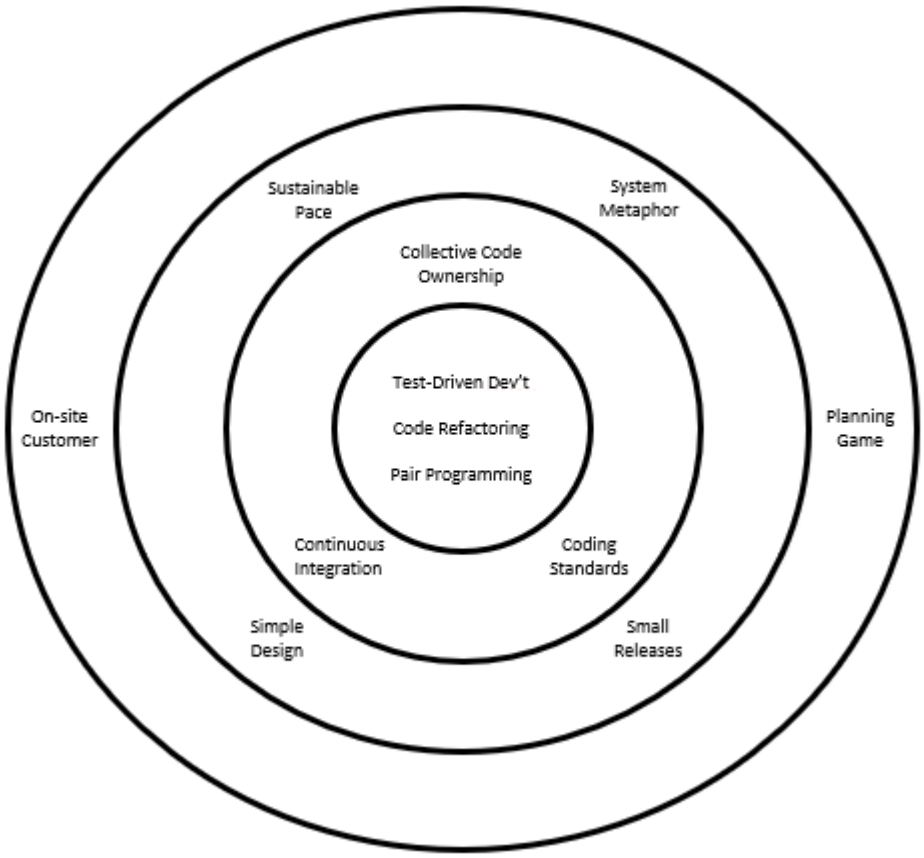
Practices. They are expressions of values that have proven helpful in improving software development. Using the values and principles of XP, teams apply appropriate XP practices in their context in their daily activities. What is excellent about these practices is they complement each other (Beck & Andres, 2004).

The old practices can be organized into categories. Among the different classifications, Meier's (2009) Four Circles of XP stands out since it groups the

practices in a layered structure (Figure 2-13) with the developer in mind at the core. These are:

- PRODUCT
  - On-site Customer
  - Planning Game
- PROCESS
  - Sustainable Pace
  - System Metaphor
  - Simple Design
  - Small Releases
- TEAM
  - Coding Standards
  - Collective Code Ownership
  - Continuous Integration
- CODING
  - Test-Driven Development
  - Code Refactoring
  - Pair Programming

Figure 2-13: *Circles of XP Practices (Meier, 2009)*



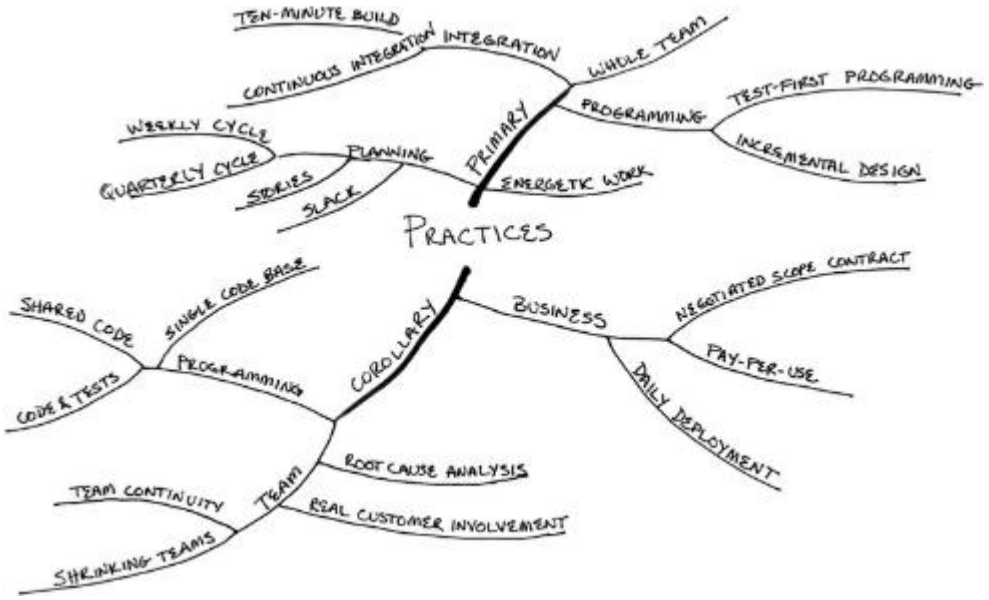
In the new version, the practices doubled in size and have two types: primary and corollary. The primary practices are independent ones that can provide immediate improvement. Once mastered, they can be extended into the corollary practices (Beck & Andres, 2004).

- PRIMARY
    - Sit Together
    - Whole Team
    - Informative Workspace
    - Energized Work
    - Pair Programming

    - Stories
    - Weekly Cycle
    - Quarterly Cycle
    - Slack
    - Ten-Minute Build
    - Continuous Integration
    - Test-First Programming
    - Incremental Design
- COROLLARY
    - Real Customer Involvement
    - Incremental Deployment
    - Team Continuity
    - Shrinking Teams
    - Root-Cause Analysis
    - Shared Code
    - Code and Tests
    - Single Code Base
    - Daily Deployment
    - Negotiated Scope Contract
    - Pay-Per-Use

Beck and Andres (2004) have mapped these practices in Figure 2-14:

Figure 2-14: *New XP practices (Beck & Andres, 2004)*



Most of the XP resources use old practices, and the developers are more familiar with them; thus, they were used in the questionnaires and interviews. The relevant old practices are Planning Game, TDD, and Small Releases (Beck, 1999).

- Planning Game - The client or business decides on the scope based on developer estimates. Developers only implement the functionalities approved by the client according to priority and schedule.
- TDD - Developers conduct unit tests; clients write functional tests. All tests should run correctly.
- Small Releases - Instead of one big release, new releases are made often and as small as possible with the most valuable requirements.

These three practices have equivalent new primary practices which are Stories, Test-First Programming (TFP), and Weekly & Quarterly Cycles, respectively (Beck & Andres, 2004).

- Stories - Stories are what drive development during planning. They are short descriptions of functionalities that are visible to the client.
- TFP - Write unit tests before the code, as opposed to the regular development cycle, where the tests are written after the code or probably not. TFP is also commonly interchanged with TDD.
- Weekly & Quarterly Cycles - Software development is planned for a week or quarter at a time. A meeting is held to manage the stories to be developed within the given cycle.

Team Roles. For an XP team to work, certain positions have a defined set of responsibilities. In the lens of the developer, he will need all the help he can get from his team to deliver his developments.

In the old XP, there are seven roles defined (Beck, 1999):

- Programmer
- Customer
- Tester
- Tracker
- Coach
- Consultant
- Big Boss

There are now ten roles in the new version, retaining only the Programmer and Tester from the original (Beck & Andres, 2004).

- Testers
- Interaction Designers

- Architects
- Project Managers
- Product Managers
- Executives
- Technical Writers
- Users
- Programmers
- Human Resources

Chapter III

METHODOLOGY

Research Design

The researcher conducted primary qualitative research, collecting data directly instead of looking for previously used data that was probably unavailable. These research results are considered unique and can open doors for developer communication.

Without a framework, variables, and hypotheses, he has adapted Grounded Theory (GT) research to collect data and has attempted to derive a theory and model after the analysis. GT applies to his research as no existing theory explains the phenomenon he is exploring. There are supporting theories, but they are potentially incomplete as the data used for them is not for the group of people that participated. Theoretical sampling, constant comparison, memo writing, and the focus on substantive theory are some of the strengths of GT that make it a good choice for this research.

As mentioned previously, he has used the CGT approach versus Classic and Straussian GT for several reasons:

- Coding Framework – Glaser and Strauss (1967) designed the Classic framework to discover a theory with systematic data analysis with a simple structure: substantive coding (consisting of open and selective coding) and theoretical coding. On the other hand, the coding structure of Straussian GT (Corbin & Strauss, 2014) is more complex in creating a new theory: open coding, axial coding, selective coding, and conditional matrix. Only Charmaz (2006) offered an open-ended

framework to construct an interpretation of the phenomenon with basic steps: initial/open coding and refocused coding.

- Philosophy – The Classic GT's (Glaser & Strauss, 1967) soft-positivism approach argues that the researcher should observe the phenomenon independently. Straussian GT (Corbin & Strauss, 2014) rejects this as its post-positivism stand states that the researcher influences what they observe and impacts the conclusion. CGT (Charmaz, 2006), with its constructivist outlook, incorporates new information for constructing knowledge rather than just passively taking pre-existing information. Charmaz's constructivist philosophy allows flexibility and considers that reality can have multiple perspectives. This reality acknowledges the diversity of the participants that are subject to the researcher's interpretation.
- Literature Review – Classic GT (Glaser & Strauss, 1967) encourages withholding the literature until the end as prior knowledge to prevent undue influences on the GT study. Straussian GT (Corbin & Strauss, 2014) takes a different position, as they recommend using literature at every research stage. CGT (Charmaz, 2006) goes much further than Strauss's, as the literature review should be used in most of the study and must have a separate compilation or chapter.

Research Instruments

The initial instrument designed for all IT practitioners was a semi-structured questionnaire (Appendix A) with open and close-ended (partial and without ordered choices) questions. The second was a questionnaire for developers using a Likert scale for closed-ended questions (Appendix B), where they rated each practice in their perceived order of importance (scale of one to five, with the latter having the highest rating). Using the principle of theoretical sampling, the researcher conducted key informant interviews with the final participants to gather follow-up information (Appendix C). The interviews were semi-structured to make the most of their time, and a research discussion guide was provided based on the questionnaire in advance. The guide questions were flexible enough to allow the participant to go beyond them.

Participants of the Study

The study participants were limited to Filipino IT professionals with at least a decade of work experience to have a degree of expertise, and they may or may not reside in the Philippines. Only 11 individuals answered out of the fifteen asked to participate in the initial questionnaire. Six have formal developer roles/responsibilities in their current or past careers (Table 3-1, where the letters A-K represent the participants). When asked if they considered themselves certain types of developers, four identified themselves as traditional, while three classified themselves as end-users, increasing our developer count to seven (Table 3-2). Most of them have used the Waterfall and Agile SDLC methodologies (nine and eight, respectively), with the former being the most used framework (Table 3-3). Although the majority (eight) have used the Scrum framework and only two have used XP at work, nine of the 11

participants (82%) have applied at least two XP practices in their profession (Table 3-4). As for the top XP practices used, Small Releases got a score of seven, followed by On-site Customer, TDD and Planning Game (six each), Coding Standards (five), and finally, Code Refactoring (four), as seen in Table 3-5.

Table 3-1: *Participants' Current and Past IT Roles/Responsibilities*

	Developer/ Programmer	Sys/Business Analyst	Func/Tech Consultant	Support	Tester	Quality Assurance	Tech/Team Lead	Project Manager	Client/ Customer
A	1		1						
B	1	1	1	1	1		1		
C	1	1		1			1		
D			1		1	1	1	1	
E		1		1				1	1
F		1	1	1			1	1	
G	1		1	1			1		
H			1	1			1	1	
I	1						1	1	
J		1	1	1	1		1		
K	1	1	1	1			1	1	

Table 3-2: *Participants' Developer Types*

	Traditional	End-User	Not a Developer
A	1		
B		1	
C	1		
D			1
E			1
F			1
G	1		
H			1
I	1		
J		1	
K		1	

Table 3-3: *Participants' SDLC Methodologies*

	Waterfall	RAD	Agile
A		1	1
B	1	1	
C		1	1
D	1		1
E	1		1
F	1		1
G	1		1
H	1		
I	1	1	1
J	1		
K	1		1

Table 3-4: *Participants' Agile Frameworks*

	Scrum	Kanban	Lean	Hybrid	XP	None/NA
A	1		1		1	
B		1				
C						1
D	1			1		
E	1			1		
F	1					
G	1					
H						1
I	1	1			1	
J	1		1			
K	1	1	1			

Table 3-5: *Participants' XP Practices*

	On-site Customer	Planning Game	Simple Design	Small Releases	Coding Standards	Continuous Integration	Test-Driven Development	Code Refactoring	None/NA
A	1	1	1	1	1		1	1	
B		1		1	1		1	1	
C									1
D	1	1				1	1		
E	1	1		1					
F	1			1					
G		1		1					
H	1			1	1		1		
I	1			1	1	1	1	1	
J		1			1		1	1	
K									1

Four developers were considered for the second questionnaire, as they have used at least four XP practices. One refused to participate further, leaving us with the

final three. The first developer, Participant 1, is a traditional one with 18 years of experience in programming and the IT industry. The second one, Participant 2, is an end-user programmer who has spent 14 years in the IT industry. He has been a development lead for four years, with around eight years of debugging, reading, and learning code. Participant 3, who has 15 years of experience in the IT industry, classifies himself as an end user nowadays. However, during the earlier years of his career, he spent ten years as a traditional developer.

The final three rated the top XP practices identified previously according to importance (Table 3-6). The On-site Customer practice was dropped from the inquiry as only two (50%) of the four developers have used it. Only the practices with an average score of at least four were selected, and they were Planning Game and Small Releases (average score: 4.333), followed by TDD (average score: 4).

Table 3-6: *Participants' Rating of XP Practices*

	Planning Game	Small Releases	Coding Standards	Test-Driven Development	Code Refactoring
Participant 1	4	4	4	5	3
Participant 2	5	4	3	5	2
Participant 3	4	5	1	2	3

Procedures for Data Collection

The initial data was collected from questionnaires created using Google Forms. Additional data was obtained through in-depth interviews via Zoom meetings, which were recorded with the participants' permission. The interviews took more than an hour each, and the researcher transcribed the video recording as verbatim as possible, with minimal Filipino words translated into English. A cleaned-up, edited

transcription was prepared for clarity and readability. As we are dealing with primary data collection, ethical measures were followed.

The researcher did not include himself in the questionnaires despite being an experienced developer. His role was to provide his insights as he conducted the interviews and analyzed the data.

Procedures for Data Analysis

The researcher conducted a qualitative approach to the responses to the questionnaires and interviews, quoting most of the data to support the data gathered. Using the GT method, data analysis alternated with data collection. Using CGT's principle of flexibility (Charmaz, 2006), he tolerated ambiguity and became receptive to creating emergent categories. He used the coding procedure of this approach consisting of initial and refocused coding, with the former breaking the transcripts into codes and the latter creating a finalized set of codes. He used a combination of inductive and deductive coding: he used literature to begin with a set of codes deductively, and once he conducted the interviews, he obtained new sets of codes inductively. The constant comparison method was conducted for the codes until he reached theoretical saturation. To aid him with coding, he used ATLAS.ti Web to help document the transcripts, organize the quotations, and manage the codes.

Initially, there were four themes produced: Practices, Roles, Modes, and Interacted Teams, and Roles with five, fourteen, three, and seven codes, respectively. Practices and Modes were lifted from the literature, while the Roles and Teams were obtained during data gathering. For the Roles theme, there were codes, which are noun forms of verbs, as Charmaz (2006) advised using gerunds to help

define what is happening in a fragment. Table 3-7 shows the initial themes and codes.

Table 3-7: *Initial Themes and Codes*

PRACTICES	ROLES		MODES	TEAMS
Planning Game	Assessing	Learning	Linear	Direct Managers
Small Releases	Collaborating	Mentoring	Interactional	Internal Developers
TDD	Coordinating	Negotiating	Transactional	External Developers
Code Refactoring	Designing	Presenting		Technical Teams
Coding Standards	Developing	Researching		Functional Resources
	Documenting	Testing		Business Team
	Interpreting	Translating		Change Management

Significant changes were made during constant comparison. Code Refactoring and Coding Standards were dropped from the Practices theme, as only the top three were considered. The Roles retained the 14 codes, but they were changed to nouns for proper identification. The Modes and Teams have been merged to form a new theme: interactions. The final set of themes and codes can be seen in Tables 3-8.

Table 3-8: *Final Themes and Codes*

<table>
<tr><th>PRACTICES</th><th colspan="2">ROLES</th><th colspan="2">INTERACTIONS (Primary)</th></tr>
<tr><td>Planning Game</td><td>Assessor</td><td>Learner</td><td rowspan="3">Linear:
- Direct Managers
- Change Management</td><td rowspan="9">Interactional:
- Business Team
- External Developers
- Technical Team
- Functional Resources</td></tr>
<tr><td>Small Releases</td><td>Collaborator</td><td>Mentor</td></tr>
<tr><td rowspan="6">TDD</td><td>Coordinator</td><td>Negotiator</td></tr>
<tr><td>Designer</td><td>Presenter</td><td rowspan="5">Transactional:
- Internal Developers</td></tr>
<tr><td>Developer</td><td>Researcher</td></tr>
<tr><td>Documenter</td><td>Tester</td></tr>
<tr><td>Interpreter</td><td>Translator</td></tr>
</table>

Chapter IV

FINDINGS AND DISCUSSION

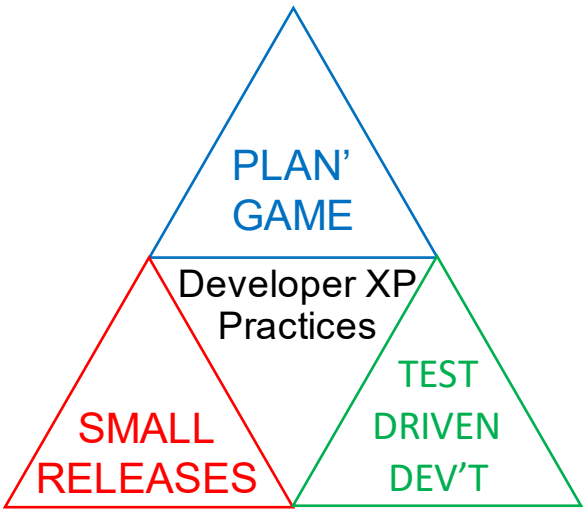
This chapter contains the detailed results and discussion of the GT study conducted to address the first three research questions:

1. Which XP practices involve the developers the most?
2. What are the different ways developers communicate in XP practices?
3. How do developers communicate in XP practices?

Developer Practices

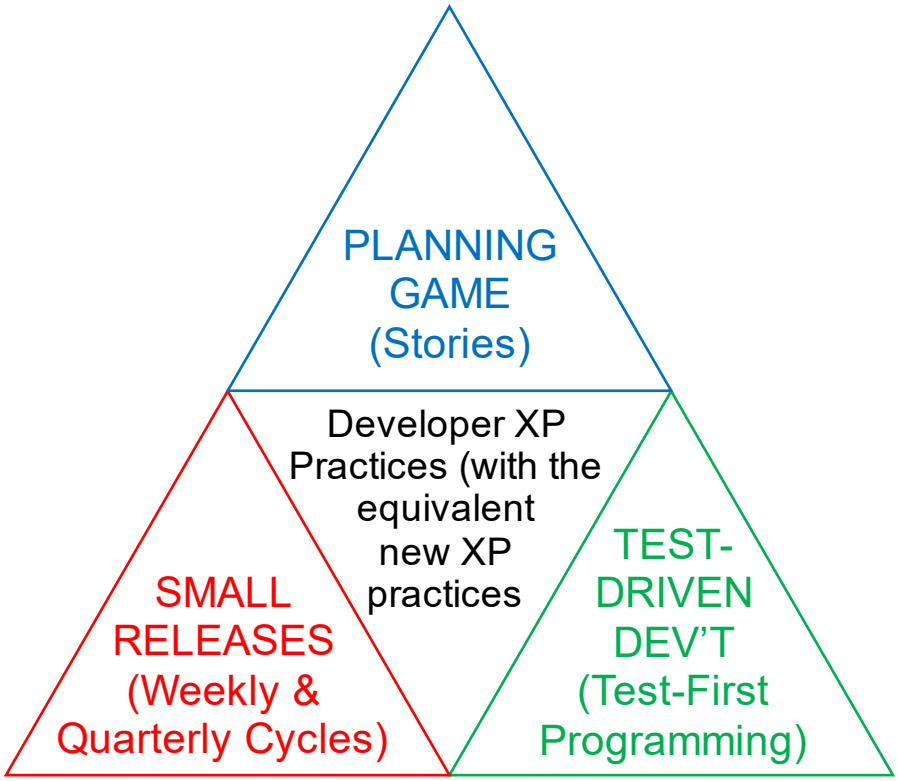
During the interviews, the researcher shared the results of the first round of XP practice ratings with the participants, and they asked them if they agreed with the results, including the low-scoring ones: Coding Standards and Code Refactoring. They all agreed, reaching a consensus. The top three practices selected by the developers in their perceived order of importance were Planning Game, TDD, and Small Releases. Figure 4-1 depicts these identified practices represented in triangles and colorfully arranged in a giant triangle. Blue, green, and red represent each practice, respectively.

Figure 4-1: *Developer XP Practices*



As the practices that came out were based on the old version, it is fitting to match them with the new one, as depicted in Figure 4-2. The equivalent primary practices of Planning Game, Small Releases, and TDD are Stories, Weekly & Quarterly Cycles, and TFP (Beck & Andres, 2004).

Figure 4-2: *Developer XP Practices with Equivalent New Practices*



Planning Game. This practice was listed in the old version (Beck, 1999) but was nowhere to be found in the new one (Beck & Andres, 2004). Perhaps the closest practice in the new edition is Stories, as far as this research is concerned.

Like any software development, this is where its life cycle begins. This project's first phase is the bond formation between the clients (or through their representatives) and the developers, so they need to establish effective communication as they depend on each other. A friendly game of tug-of-war begins: the clients decide on the scope based on the developers' assessment. At the same time, the latter only implements the approved stories or functionalities of the former.

According to Participant 2, the aligned scope of the planning game will make everyone happy, including the business, while having an efficient process if done correctly:

> *The strength of the planning game is that we can give benefits to the business faster and in small chunks. It's like a 10-story house that I want. Rather than waiting for so long to get it, let's do it by level. Businesses will appreciate it more, and they will get the benefits faster.*

Participant 3 agrees with this; it is good because it is defined. If the business can lock in a proper set of requirements, then the developers can design it clearly:

> *If the scope is done well, the surprises will be minimal. So, we can expect to have a definite timeline. We can reassure them we can finish it in a month or so; it can be a bit more predictable in that case. It all depends on the quality of the requirements and the design. It depends on how good the visibility is between how we developers imagine things to be and how things really are.*

However, most requirements are flexible; they sometimes change during development. Thus, the burden of having reasonable requirements or stories relies

not only on the client but on the knowledge of the developers as well. As Participant 1 puts it:

> *Your strength is your knowledge and experience of the solution you support. If you are more knowledgeable and confident in giving definite answers, you are more aware of the solution's limitations and capabilities.*

Sometimes, the developers must familiarize themselves with the solution so that planning can be complex. Participant 1 continues:

> *Sometimes, you are put into a spot where you do not know if it will work. That is a difficult situation for us. If there is a new solution that we do not understand, we look into it. We do not commit. We are not saying it will work, but we will get back to them. That is the usual answer we give if we are not 100% sure.*

Therefore, discussions are essential between the client/functional and development teams to decide if the scope is feasible, especially the complex items. There are consequences, according to Participant 2:

> *If we don't do proper assessments during the planning game, we will either overcommit and then definitely not meet expectations, or we will have overtime and overload happen to the developers as well.*

TDD. The related practice mentioned in the old version is Testing (Beck, 1999), which TFP replaced in the new version (Beck & Andres, 2004). Although Beck has not explicitly mentioned this as a practice, he is also credited with having rediscovered TDD, a technique that encourages simple designs (Beck, 2002). TDD has become popular outside of XP, with numerous research papers mentioning it.

Testing is crucial prior to the coding of the solution. The clients write functional tests, the developers conduct unit tests at the minimum, and the users run acceptance tests (UAT). Effective communication is needed to ensure that all tests run correctly. As Participant 1 shared, they let the functional team do the full testing, with minimal unit tests on their end:

> *I do not want to add more testing on my end. At a minimum, we do what was requested based on the test. There will be some issues, and you expected this, but the requirement does not cover it. So, sometimes you see a scenario that fails in this area, but sometimes, functionally, you are not fully aware (that it is not).*

The unit tests are equally important to the functional tests and ideally should be prepared early, as attested by Participant 3:

> *Normally, we develop the test cases after we do the design because we already have a clear picture of how the change would look—or at least. Ideally, we should also have a clear picture of how to test it. When the development finishes, we execute the tests.*

Participant 3 recalls that during UAT and other tests performed by the developers, there were unexpected scenarios not accounted for in the design:

> *Normally, our testing is more on the unit and site integration tests. So, we catch a few bugs there, where the program gets refined. That's where the program really gets working based on what we expect the scenarios to be. And then, during the UAT, scenarios not discussed before come in. Normally, surprises come in most cases during the actual user acceptance testing. So, when we have almost like production, like data coming in, and working with the solution, because that is where all the funky combinations*

come in when it was a controlled environment when we were doing the test cases, we might find some bugs, but they are not surprises. They are just like wrong coding that needs fixing, which is normal. However, the surprise nearly came during UAT, which was late in the game. So, then we must scramble.

During development, it will be advantageous for developers to have functional knowledge as well, according to Participant 1:

Having functional knowledge and understanding of what you are doing is good because you can foresee additional requirements before the actual developments. If you are purely a developer, you will rely on the documents they provided with the requirements. However, if you have functional knowledge, you understand the difficulty or how easy it is to do the development. So, you will also know your way around doing some of the tests.

It sets expectations for both the functional and developer because the functional will now explain fewer technical details, and you will also be more interested in the requirements rather than the functionality of how things work.

Participant 3, on the other hand, has functional persons who also have technical know-how, which makes development easier:

In our environment, business users do not interact directly with developers because developers usually do not yet grasp what the report is about and what it is supposed to be doing. So normally, we have an in-between, a functional person who also knows the technical side and understands what the business needs and why they want it. Then, they can translate that into what needs to happen. Technically, what needs to be changed in the program to make what the users want? So, they translate that and then pass it on to

the developers who make the actual changes in the system. They drill down to the action.

Small Releases. The old version included this in the earlier practices (Beck, 1999). It disappeared and somehow changed into primary practices, Weekly and Quarterly Cycles later (Beck & Andres, 2004). There is even a corollary practice, Daily Deployment, from the new version. This practice was shown by Participant 2 in exceptional cases:

Daily is really for the impact of day-to-day operations; we need to act on it. We cannot let the business or manufacturing stop due to an issue. It is not an official plan to release daily, only if there are urgent, emergency cases.

Some companies (like Participant 2's) also have monthly releases instead of quarterly. New releases are scheduled more often instead of having a big one.

One advantage of having Small Releases is the quick turnaround time, according to Participant 1:

Sometimes, the expected changes are small or only bug fixes, so they're easy to handle. A lot of small changes are coming, but if there are smaller releases, they are easy to add, especially when they are just minor tweaks or something like that.

With more minor releases, it is easier to steer the developments in the direction the business wants. Participant 3 has this to say:

It is very powerful when it comes to tweaking and ensuring that what gets developed is what is needed and what is wanted. Sometimes, the users want something, and then you develop it, and that entire process takes months. However, it is not really what they want when it is live. However, it is in

production already, so there is nothing to do unless they make another change. Here, it is easier to see where the program is headed.

Managing the small releases becomes a problem if not handled properly, says Participant 1:

Sometimes, although they are small changes, there are a lot of them that you must manage. So, aside from making the changes, there is a lot of administrative management and active work involved. That is one of the challenges. We are always worried that the volume will increase with the number of small releases coming together.

Participant 1 laments that when a massive project gets split into minor releases, there is a tendency to have backlogs:

Although tagged as small releases, backlogs need to be implemented in sequence, and there are dependencies. So, you must be able to complete one of the changes to move forward. This is one of the challenges for small businesses because there's a short period of time that you must implement something that has a dependency on the neck.

When the window is too small, say weekly or daily, there are many risks, as relayed by Participant 2:

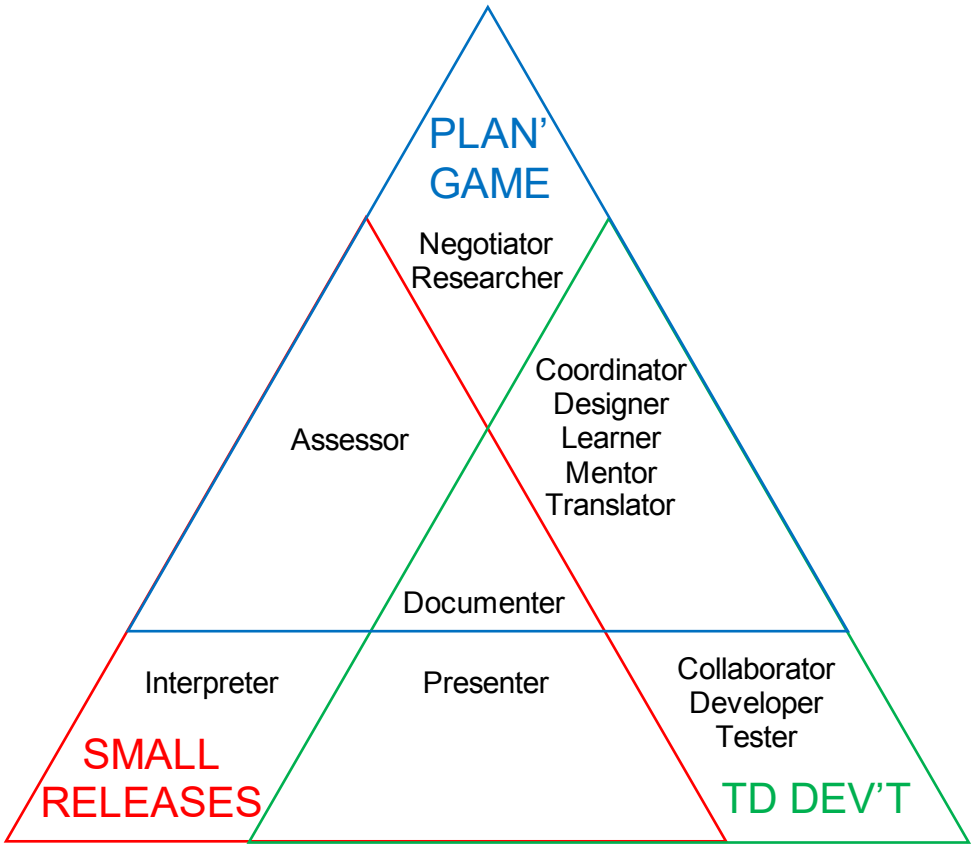
A lot of risks come from the weekly releases because the business is saying it is urgent. Everyone is rushing through it to develop it, run all sorts of tests, and move it to production in three or four days. So, if we are looking at that type of timeline, something will be missing. Everyone is trying to get it done because there is an issue in production. It is an urgent emergency case, so the risk will always be high.

Developer Roles

In the new XP version (Beck & Andres, 2004), programmers estimate stories, break them into tasks, and write tests and code to improve the system. These roles seem limited, but Beck also states that a mature XP team is not fixed and rigid. Everyone can do his best to make the team successful by suggesting improvements, keeping in mind the alignment of authority and responsibility. A developer may be knowledgeable about infrastructure and can suggest architectural improvements if they can, perhaps depending on their seniority and influence.

As mentioned in the previous chapter, the study identified fourteen roles: Assessor, Collaborator, Coordinator, Designer, Developer, Documenter, Interpreter, Learner, Mentor, Negotiator, Presenter, Researcher, Tester, and Translator. In Figure 4-3, practices overlap to show how developers communicate under specific roles identified per practice.

Figure 4-3: *XP Developer Roles*



The seven groups that emerged according to practices are as follows:

- Planning Game only: Negotiator and Researcher
- TDD only: Collaborator, Developer, and Tester
- Small Releases only: Interpreter
- Both Planning Game and TDD: Coordinator, Designer, Learner, Mentor, and Translator
- Both Planning Game and Small Releases: Assessor
- Both TDD and Small Releases: Presenter
- All three: Documenter

It's important to note that the roles within each group are independent of each other. This means that the roles that appear in the same group are not necessarily related to one another, highlighting the non-hierarchical nature of these roles in XP practices.

Based on observation and the researcher's experience, the roles follow certain types of communication, albeit some are optional: verbal, written, and visual. These will be discussed in detail below.

Planning Game only:

Negotiator. Deals with the prioritization of tasks. Suggests alternate solutions if development cannot be done or there is another way.

Participant 3 negotiates during planning, especially for complex solutions that require more time:

It was also a negotiation during those (planning) calls because if they wanted that (solution), that would take more time. What I can offer is this one that gets the same job done—it's just not that nice, but it works faster. So maybe that

works, and then if everyone agrees, we can proceed with that simplified version for the sake of time and being able to work on other topics.

It is part of the planning stage where we put buffers in there. Because we've had that instance before when we didn't have buffers, we really must negotiate new timelines because it's not workable. Everybody ends up reworking everything, so it is a delay.

Participant 2 uses a tool to manage his team's workload and prioritization of tasks:

Right now, we are trying to standardize our intake process. This is critical for us because we are using one tool. We are trying to define that process and how they (functional) can raise tickets or requests to the dev team. This is currently important for us to be able to manage the workload and prioritize the tasks. We try to avoid engaging via email and chat because that will get lost in translation.

As a development lead, Participant 2 is concerned with his team and negotiates to make sure that they do not get overloaded and push back whenever possible:

Most of the time, we also try not to overload the resources. However, there will be times when the development team cannot do much because of urgent tasks, a production issue, and a tight project timeline. It is more of a case-to-case basis that we try to be flexible.

We try to (be flexible) and not push back; it is more on having options given to them (functional) rather than just saying no. So that is what we are trying to practice as well. If we cannot do this, they will understand that maybe in the next release, they will get what they are requesting.

So, we chat with them or inform them that this is X capacity or, based on the change, it will only meet the three-day requirement if it is just a one-line code change. If it has a significant impact, we will make an exception. If not, we will push it back a week to give us some leeway because things might go wrong. Even if it is a one-line change, it might impact others. It looks simple to them, but if no proper assessment is done, it might also have a big impact.

Participant 2's team conducts handover sessions, and he aspires that his senior developers will mentor the new ones as they grow:

We do handover sessions for critical processes that they are supporting. We would have specific calls, and maybe two developers would join that call. We have two developers who can interact and maybe try to understand the process. What are the interfaces and integration points? Those are the main ones, but the majority would be able to go through it and understand what is happening by reading the code, so it is one of the integration points in our critical business processes.

The team is relatively small right now, but once the team grows, we will see that there will also be functional experts. Based on those functional experts, they can mentor the juniors so that it is more of how it goes.

The negotiator role has the planning calls for verbal communication. As for written communication, the task details and prioritization can be found in formal tools such as ticketing systems or informal channels such as email. The alternative solutions can be illustrated as part of visual communication.

Researcher. Investigate anything related to the solution, such as functional and technical knowledge. Earlier, it was mentioned that a developer may not be sure

or knowledgeable about the solution. Although we may think it is the functional person's responsibility to make the developer understand and give more details on the functionality, sometimes they are unaware or unsure. For Participant 1, researching is a task for both the developer and the functional.

> *The functional (team) needs to provide more details, and the developer should start checking if it is possible with the data that you have. With additional information coming in, you may have a better idea of what the requirements will be in the future.*

For Participant 2, the developers and the functional team are partners that support each other. When the former needs time to research a possible solution (e.g., proof of concept or POC), the latter gives them time to do so:

> *Yes, that is part of the assessment. They (the functional team) would be open if this were more of a POC because we are not sure about the solution yet. So, if we talk about a POC, we will do it in the sandbox and allot more time to it.*

The developer, as a researcher, can have functional and/or technical consultation meetings for verbal communication. The research materials cover written communication. Research images, such as screenshots, are part of visual communication.

TDD only:

Collaborator. Work together with other development teams, internal or external. This role differs from one of the XP practices, Pair Programming, which involves sitting together on one machine (Beck, 1999). These can involve developers who initially worked separately but must collaborate on their developments for end-

to-end testing, like Participant 2, who shares his experiences with external developers and functional resources:

> *In terms of sharing, there would be collaboration from their (internal and external developers) end, especially if things are not working. I am also trying to encourage the teams, even though they are working on different functions. There can be things that can be shared with others that they can also use in their respective functions. That is being encouraged.*
>
> *Regarding testing (with the functional), they do the testing with the developers in the unit test.*

The collaborator role can have cooperative meetings for verbal communication. Both code and text are involved in written communication. Diagrams and end-to-end flows can be part of visual communication.

Developer. Also called programmers, they are the heart of XP (Beck, 1999) and are tasked to develop programs. They work with the functional resources to deliver requirements.

Previously, we mentioned coding standards where the participants had opposing stands. For Participant 3, there should not be any strict rules on them. Fortunately, some tools can aid them. He has a lot more to say regarding this topic:

> *We do have to follow them (coding standards), but I do not see much benefit in them from a developer's point of view. You must follow them just because the customer needs them—it is more of a matter of because we need to.*
>
> *There would still be difficulties, even if they (developers) followed the coding standards, because everyone has their coding style. So, the coding standards do not help with that. Whoever is taking over still needs to get used to the*

style and read it from a different point of view. I saw that the coding standards do not really help with that. They help with understanding what certain variables are for if they are global variables. If it is a local one, that is where they can help. However, as long as there is a proper description, it is good enough.

We do have some automated tools which check the coding standards. So, they automatically read the program code and check if anything is not aligned with the coding standards. That helps with making sure that those coding standards are followed. It is not beneficial to the developers; it is more of a regulatory requirement. However, for me, what helps from a developer's point of view is that the variables used make sense with what they are being used for, like just putting an X or a Y. They are naming it correctly, what it stands for.

As multiple tests are involved, the development can still go back and forth between the developers and testers. Bug fixes and missed scenarios are some of these "surprises" than factor recoding, as mentioned by Participant 3:

Normally, they (developers) have the unit and site integration tests. Those are mostly still within the functional and technical teams. We do catch a couple of bugs there, and that is where the program gets refined. That is where the program gets working based on what we expect the scenarios to be. During the UAT, the scenarios that were not discussed came in.

As the coding process is usually an individual task, the developer role only has verbal communication if two or more programmers are involved. In this case, formal or informal meetings are present. As for the written communication, code and

code comments are involved. The output of the program(s) involved, if any, is considered visual communication.

Tester. The person with this role ensures the solution works according to the agreed-upon requirements. Developers are tasked with unit testing before handing the development over to functional resources or the business.

According to Beck (1999), development is driven by tests, which is the concept behind TDD. The test cases should be built first; Participant 3 attests to this:

> *Normally, we design test cases before coding. We put the developer test cases after we do the design because we already have a clear, or at least ideally should, a clear picture of how the change would look. If we have a clear picture of what it looks like, then we should also have a clear picture of how to test it. So, usually, when development finishes, we execute the tests. Often, the test cases move. So, we have an initial list of test cases, and then, as we develop and progress with the program, we add or modify test cases. Then, when surprises come in, those get modified again. It is almost never the same version as how it felt in the beginning.*

Some of the challenges Participant 1 must face are the availability of test data, presentation of test results, and communicating with the business if the tests cannot be done:

> *Sometimes, we cannot foresee the possibilities in the system. It seems possible, but later, we discover that some data is missing or unavailable in the system. Then, we cannot proceed. So, it happens before and even after developments.*

Aside from getting the correct test samples, the challenge is capturing the results. Sometimes, you take many screenshots and miss something, making the test irrelevant. Sometimes, as a developer, you do not know what the screens or data they (business) want to see. Sometimes, you forget to include that in their documents, so you must retest again. Usually, the challenges we encounter in testing are simple test scripts that require you to know what data to put in and what they want to see.

You communicate with the business that this particular test is not possible. So, it will be a business decision; then, the test will be flagged as not applicable because, technically, it is not feasible. So, you discuss it with the business and review all the test cases to see how many have passed and failed. This change must be based on the current settings when communicating with the business. They decide whether to create new tests or mark them as not applicable to the current test cycle.

Aside from unit testing, other technical tests may be required from the developers, such as Participant 3's regression testing, to ensure that the new changes do not affect the system. He also mentions about automated testing that will be helpful to the developers:

We do regression testing for bigger changes. We also see an impact even for smaller changes because even a one-liner code change can impact the business. We are aware of the impact and the risks that we are taking.

We do unit testing, but developers can only do so much in terms of unit testing, right? So, when you get to the functional, they say a process was missed. How would the developer know? Automated testing would come here because all processes should be there. Everyone is focused on building the

scripts inside that tool, and then the developer can run everything and wait a few hours to finish everything; the reaction time will be faster. So having that automated testing or maybe it is a team that is handling that will make things flow smoothly and faster.

The tester's role involves meetings for the test plans, including test cases and UAT scenarios for verbal communication. The test details, along with other types of assessments such as UT, SIT, and regression tests, are documented for written communication. The presentation of test results can include screenshots as part of visual communication.

Small Releases only:

Interpreter. Usually, for the business, developers translate the solution or code to functional logic or flow. According to Participant 1, the business is not usually familiar with technical terms, so the changes need to be interpreted more simply:

Usually, since they are not that technical, the problem or issue in communicating is how to simplify the changes so that they will understand them better. Sometimes, you make some changes that are purely technical, and the challenge will be how to explain to them in a non-technical way what the changes or improvements that you have implemented were.

Sometimes, a developer interprets by showing and telling; for instance, Participant 1 must do this to prove that the solution is not possible:

There are some instances where you must do it. As I said, they are (business), not technical people. So, trying to explain to them technical stuff, they do not appreciate it. They do not appreciate or understand what you are trying to say. The best way to show them that it does not work is to do it to get

their confidence. You can explain it to them, but they really cannot understand why it is not possible that you must build it and then prove it is not working.

The interpreter role involves "show-and-tell" sessions for verbal communication. For written communication, interpretation contains functional logic and non-technical text. A presentation that usually shows the flow of the program or solution can be considered visual communication.

Both Planning Game and TDD:

Coordinator. Aligns and synchronizes developments across affected technology teams. Depending on their needs, the developers work separately with various teams to achieve their goals. For example, Participant 2 is working on back-end development. Upon completion, he coordinates with the front-end developers. It can also be the other way around, where he asks the front end if they are done so they can pull their change from the back.

Participant 1 works with the technical and functional teams from planning until testing:

If parts of the change cross different roles, you do not just require a simple program. For example, you must coordinate and work with other IT resources if you need to pull data from somewhere. So, it depends on the scope or the magnitude of your change.

If we cannot test it (the change) as a developer, we pass it to the functional team. We do the unit tests, but if it is impossible for the developer or there are many functional settings required to do the test, then we pass it to the functional team because we are not equipped with the data required to do the complete unit testing. For example, they want something to change in the

sales order processing, but setting up the sales parameters usually, we are not aware of it. So, we pass that test scenario to the functional.

The coordinator's role can be to hold alignment meetings for verbal communication to synchronize developments. As for written communication, the relevant development details, such as input and output needed by the technology teams, are included here. The handover documents can also include diagrams as part of visual communication.

Designer. Define the technical components of the requirement where the result is a technical solution. The requirements should be well-written to achieve a successful design. That's why for Participant 3, these should be adequately defined in the planning session; otherwise, there will be "surprises":

It is good because it is defined. If we can lock in a proper set of requirements, we can design it clearly. If the scope is done well, the surprises will be minimal. So, we can expect a definite timeline. We can reassure them that we can finish it in a month or so, and then it will be more predictable in that case, but it all depends on the quality of the requirements and the design. It depends on how we imagine things to be and how things are.

The main challenges we have encountered are about the surprises. Because we plan something, we know what we want to do. However, depending on the complexity, sometimes you are in the middle or almost finished with the development. Then, there is a bug or a scenario where you have not considered it. It is not something you have planned for. In that case, it is a small amount of free work because of this surprise or this new scenario; you have to plan for it again. Then, I designed, developed, and tested it again. It is

not the smoothest thing, but we usually have a workaround: we always add a buffer when working on timelines. There is always a buffer for surprises because the surprises are often not so surprising anymore as they are factored in.

He continues that sometimes, due to the rework, some designs are just band-aid fixes to make the solution work:

It is the rework. Because of that, you will have to pay for it when you do not capture all the requirements initially, especially with more complex changes. You have to go back, develop, and design again. Sometimes, those designs are more of a band-aid fix. Sometimes, the entire design does not go well because it just must work, and you cannot redesign it anymore. It is not always the prettiest design afterward.

The designer role has functional calls for verbal communication, especially when the developer needs clarification. As for written communication, functional requirements, and technical solutions are some documentations created in this role. The user interface (UI) and user experience (UX) designs are part of visual communication.

Learner. As a student, the developer gets trained in functional or technical skills relevant to the project.

For the former, Participant 2 states that the functional resources will share the concepts regarding functionality or any new features yet to be known to the developers, but they might need to learn the technical details. They might provide a training session if needed, like what Participant 1 echoed:

Usually, they provide sessions. There's user training. They explain what is happening, what the solution is, and what we want to achieve. So, you get on board with the solution itself. You appreciate more what the requirements are and why you are doing the development. The training is also for the project team, so everyone is aligned with the primary goal.

There are also times when the developer, such as Participant 3, approaches other developers, internal or external, for suggestions or ideas:

Normally, if that is needed, I would share the requirements with them (developers). For example, the business requires a report that needs a grid, but it must be modifiable. If we could go with this approach, they would say you can use it since it is newer or better. However, usually, that is how I would approach them. They would not get the entire requirement, but they would get that one I am consulting with them about, which is the area where I am unsure what could be done best.

The more senior ones (developers) often have a different idea. That is great because they often know more than I do. So, I am very much interested in listening to that, but it highlights their experience if they're not that knowledgeable yet and they just go with what I say. They know better than they speak up, which is great. So, either way, we get my default idea, or they come up with something better. That's great in that clinic.

There are instances where I am not sure how to implement something technically. So, I tell them (developers) the high-level (requirements). What I want to happen is that this data must be transformed and split into two different things, for example, but I am not sure how that works. Then, I ask them for ideas; often, they would have an idea or two.

For optimization tasks or instances where the developer wants to improve the code's performance, they either go back to the original programmer or consult others. Here is what Participant 3 has to say:

> *There are some instances where we have to go back to the original developer. It does not happen often because they often will not remember so much anymore. However, it is often up to the person optimizing the code to understand it. If they do not, they piece it together with the functional guys. So, they give a best guess and then see if that works during testing.*
>
> *The person who is doing the optimization can consult with others. Can you have a look at this? Do you think we can improve? Then, he has several ideas. At the end of the day, there's only one person who takes care of coding the changes to prevent possible conflicts with merging the versions or conflicts with the logic.*
>
> *That different person (developer) would have a different point of view on the program, and they would be able to see better the gaps in that program and what could be improved. Usually, we find more improvements that way than if the same person looks at it because they programmed it that way. So, if they look one more time, they may find some things (to improve), but there will be fewer.*

The learner role may have technical/functional training sessions or consultation for verbal communication. For written communication, these can be high-level requirements and code snippets. Suggestions and ideas, when drawn, can be part of visual communication.

Mentor. Acts as a technical advisor, usually for senior developers who can be considered subject matter experts.

For Participant 1, the mentor steps in when a new developer takes over his previous code, providing a clear understanding and guidance:

> *Usually, it happens when the project has already ended, but in the future. They started, or they want to implement other solutions. Sometimes, I get involved because I was the original developer. So, I need to explain to the new developer what was implemented: the requirements and the approach we took in this part of development. I explain so that they get on board more quickly with what was implemented or the solution with the client.*
>
> *The expectation is that they (new developers) know the code; they just need to understand the flow. That is why not at the code level, but you explain how it works so, at that level, the objects and tables that you use, but not the exact coding.*

The mentor role involves informal or formal advising sessions and consultations for verbal communication. For written communication, onboarding documents and code can be used for discussion. Diagrams such as flow, objects, and tables are included as part of visual communication to aid in explanation.

Translator. Developers with this role convert the requirements or logic into code or pseudocode. Most of the time, the developers translate the functional documents into technical specifications. For Participant 1, this also involves placing pseudocode for their documentation:

> *It (technical document) does not contain the exact coding itself, but the logic and the high-level idea or the gist of what will be needed is there. Also, we do*

not put the entire code in the technical specifications, but we put in an idea of what will be created for the technical document. Not the details, usually, just to give an idea of the development or if the logic is aligned with what is expected or what is requested by the business.

The translator's role includes optional functional consultations for verbal communication. For written communication, the technical specifications translated from the functional documents contain code and/or pseudocode. As the translation process involves mostly code, only the (math) symbols used can be considered part of visual communication.

Both Planning Game and Small Releases:

Assessor. Assessment involves gauging the difficulty of solutions and providing effort estimates. The functional resources initially assess the requests from the business, but sometimes they need help from the developers, like in the case of Participant 2:

The business would raise the request, and the functional team would assess. Is this a big change? If they are not sure, they coordinate offline first with the developers. Can you help me assess this part of the change? Is it a small one? With that, it's more of an aligned approach that we can meet in the current or next cycle.

Once the requests have been cleared from the functional side and handed over to the developers, they undergo a thorough assessment for completeness. This process is similar to what Participant 2, a team lead, and his sub-lead follow:

I have a sub-lead who helps me day by day. I also need the sub-lead to assess the tickets coming in to ensure we have quality specs. Then that is

also where we try to understand if this will fit in with the current cycle because some of the requests come in, like I need the change in three days. We must go through unit tests, UAT, and all, so three days are impossible. That is also where we try to delimit, and the sub-lead is helping me do that.

As a seasoned techno-functional developer, Participant 3 has experienced giving effort estimates to developments as part of the technical team:

When the requirement is presented from the business side, they discuss amongst themselves which one they want to pursue. When they've decided on a particular change, they approach the technical team, who will then make effort estimates. Based on the effort, budget, or time that it takes, they decide on a scope. Then, they assign a designer and then a developer.

Often, the other functional folks will have an idea of what they want. As the developer or the guy who knows the technical side of things, I know it takes too much time and effort. Then, we must discuss what is more practical so we can decide to simplify something.

Prioritization is a must for Participant 1, either based on urgency or a first come, first served basis:

In one project that I encountered, we had sprints. We prioritized the developments based on what they needed, or which were simpler to implement.

If the project is staggered or in phases, you will have small releases. For example, you have a solution, and the development is sometimes part of a chain of changes you must apply, so you prioritize which comes first. That is why you do small releases: to accomplish certain things at a certain level. You complete certain things daily or weekly instead of waiting for everything to be

completed before moving the changes into production. They want many change requests but perceived future FTE (full-time equivalent) or developer resources are limited, so you prioritize.

The assessor role has assessment meetings between the developers and the functional folks for verbal communication. As for written communication, the estimates are included in the task details in ticketing tools or other tracking systems. Anything illustrative that can aid in assessing the solution is part of visual communication.

Both TDD and Small Releases:

Presenter. In this role, a developer discusses the solution using visual tools. Participant 3, for example, is fond of showing his screen and drawing something in Excel, especially when it has specific requirements. He tries to draw what he envisions the development and output would look like. Then, he asks for ideas and how to implement the solution best, depending on what the user needs.

Participant 1 prefers to present in a more visual way using flowcharts when explaining the solution:

Usually, the presentation data flows, and it is much easier for them to understand, for example, from A going to B, C going to D, why A is not going to B, and why A is going to C. It is easier for them (the business) to visually appreciate the change in data flow or the process. Although we submit some documents with functional and technical specifications, we usually present them more visually because sometimes they also do not understand written documents with technical jargon. So, it is the presentation and data flows that they appreciate most.

For presenting the test results, this participant uses screenshots:

These (test results) are documented with screenshots, but these are usually presented during the testing results call, especially for projects. For projects, they usually must pass a certain number of test scenarios with the business. So, the tests are discussed with the people concerned, and we present them with screenshots.

The presenter role has the business presentation for verbal communication. As for the written communication, the solution and the test results are exhibited with minimal technical jargon. The demonstration for the business is expected to have flowcharts and screenshots as part of visual communication.

All three:

Documenter. Records important information regarding the solution, usually involving technical data.

Documents are created at all stages of the project, starting with planning. Participant 1 describes his role:

We create the documents while we plan. So, usually, we have a high-level design, but it's not the formal document itself. However, during planning, you also get information from other areas, so you collate that into your planning document.

When development comes, all the participants put comments in their respective codes as a form of documentation:

Usually, by practice, I put a note on it (code). For example, this subroutine will do this; this function is doing this. I put them (comments) on certain parts of the code and explain a little of what is happening or what will happen with

the whole process flow. I do it to follow the standards and help me understand in the future. Because in the future, I tend to forget, but at least I have a guide to what is easier for me to follow. – Participant 1

They (developers) put comments on the code. We have been discussing documentation. Do we want to do technical specifications? I was asking them what they would put in the Word document. I said parcels of code and then explained it again. If that is the case, no one will read that unless it is an audit requirement. If it is not an audit requirement, no one would read that. Developers would go to the code directly rather than read a Word document. So, we are trying to decide which way to go, or us, regarding technical specifications. Because if no one will read it, then why do it? Let us put it in the code or maybe do another class to put all your documentation inside so everyone can go to that and understand what is happening within the code. So, I would say that it is way more efficient in-code documentation. – Participant 2

Normally, I like putting the comments in the code, like the pseudocode. If there is a chunk of code I am putting in, I try to comment on what that chunk does just because I would forget eventually. In addition, we have these technical documents that we complete, but often, these are not as detailed as just reading the code itself. So, I still look at the code itself for the complete picture. – Participant 3

Despite the presence of code comments and coding standards, technical documents remain a crucial resource, as Participant 1 rightly pointed out. They provide a reliable source of information, ensuring that nothing is missed or misunderstood.

Well, you still need the (technical) documents because they're one area where you can access all the information or the changes. If you read the code, it might take you some time because it might be huge. It is easier to understand the document sometimes. So, if you follow the code, it will help you, but it's not a good idea to read it. It will take a lot of time, especially if you're not familiar with it.

The complete code is usually not in the documents. Snippets of it, the pseudocode, or something similar are included. It is to give an overview of what needs to be done or what is happening, but usually, it is not the exact code itself.

Test plan documents come from the developers as well, Participant 1 continues:

The test plan of the document usually includes the coding and what areas to test. So, you identify only the changes that need to be tested. But sometimes, the actual test scenarios and who will test are not yet included during the planning stages.

We still include screenshots for simple testing, but sometimes, it is just a simple message that something was created. However, the value within it, the details, the quantity, and the materials are not included in the screenshots. Sometimes, they (functional) require it, but it's not fully stated in the test documents that you need that information as part of the results.

For Participant 2's Change Advisory Board (CAB), these test documents are far more critical than the others:

Right now, testing is more important for the CAB. That is why if we miss something there, if you don't have the right scripts, it has a bigger impact. So,

for now, I guess technical specifications wouldn't be looked at because they don't have much impact.

The documenter role has optional meetings for verbal communication. As for the written communication, we have the technical documents, standardized code, pseudocode, and code comments. Any illustrative parts, including code snippets, are part of visual communication.

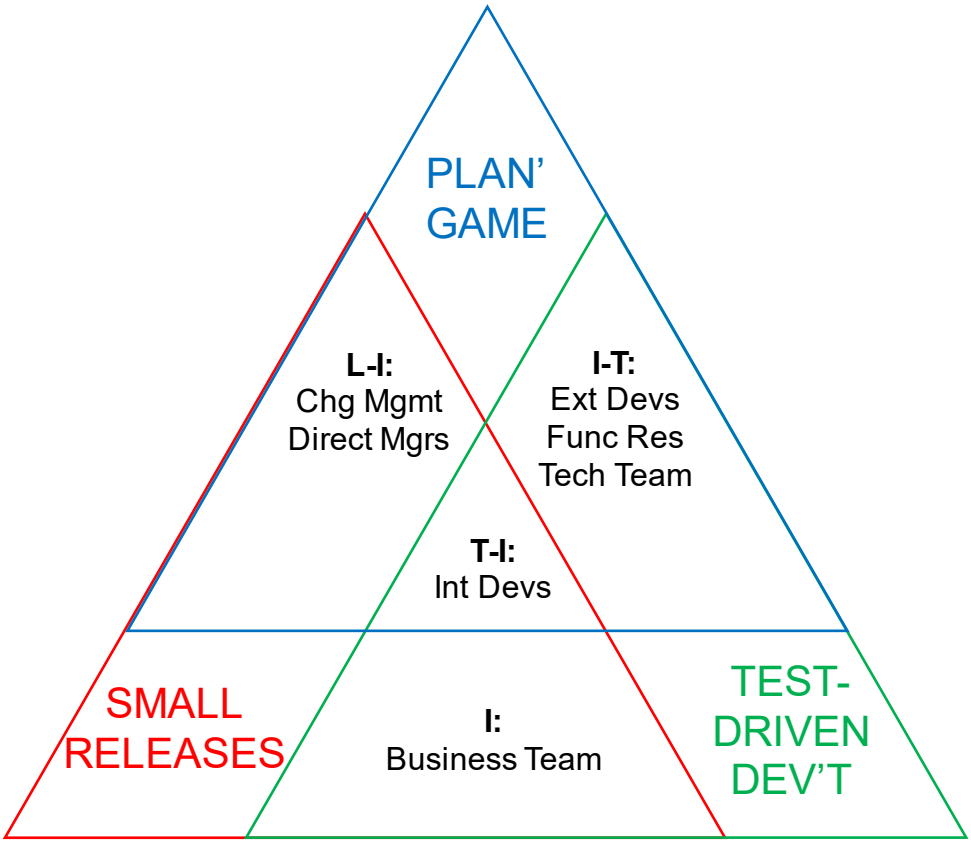
Developer Interactions

In the new version (Beck & Andres, 2004), nine members are involved in an XP team, as mentioned in Chapter 2. This research identified seven teams interacting with the developers in the codes: Business Team, Change Management, Direct Managers, External Developers, Functional Resources, Internal Developers, and Technical Team. Additionally, these teams are divided further into four groups of interactions derived from the models of communication discussed in Chapter 2: Linear, Interactional, and Transactional.

- Linear-Interactional (L-I): The primary mode is Linear, while Interactional is secondary. These include the Change Management and Direct Managers.
- Interactional (I): The primary mode is Interactional, with the Business team as the sole member.
- Interactional-Transactional (I-T): The primary and secondary modes are Interactional and Transactional. This group includes External Developers, Functional Resources, and the Technical Team.
- Transactional-Interactional (T-I): Here, the Transactional mode takes precedence over the Interactional mode, and only the Internal Developers are involved.

We have again used an overlapped version of Figure 4-1 to reflect the developers' interactions with the represented teams according to practice (Figure 4-4). The modes of communication are in bold letters.

Figure 4-4: *XP Teams that interact with the Developer*



Six teams were present in two out of three practices, while the remaining team existed in all three. The four groupings according to interaction were:

- Planning Game and TDD: External Developers, Functional Resources, and Technical Team
- Planning Game and Small Releases: Change Management Team and Direct Managers
- TDD and Small Releases: Business Team
- All three: Internal Developers

Linear-Interactional:

Direct Managers. This group comprises project and product managers, who are mentioned as members of Beck and Andres' new XP practices (2004). Technical ones, such as development managers, are also identified here, which can be equivalent to Executives in Beck's who directly influence the developers. It seems their equivalent role in the old XP version is the Big Boss, and they can be Trackers or Coaches as well (Beck, 1999).

As Participant 2 has a Development Lead role, he acts as the gatekeeper of the developers during Planning Game and has some supporting statements for this section:

> *If the timeline is fixed, we check if there is too much to accomplish. If yes, we would ask for a budget to augment the staff of developers or other people. So that is the only time they want us to finish this by X date; we would need X resources. If you do not have a budget, then delay the project. It is one or the other.*

In Beck's definition of Project and Product Managers (2004), they both facilitate communication inside the team. The only difference is that the former coordinates with outside entities such as suppliers and the rest of the organization. It is vital as they are the ones handling the prioritization of the tasks during planning:

> *All requests are in our queue. If there is too much for the development team to manage, we go back to the functional leads that we have this number of requests. Let's say, for example, we have five requests, and we only have this much X capacity. So, which one do we prioritize? Which one can we delay? That is more of the partnership and to plan out what will be the business impact and what we can delay as a result.*

Right now, we prioritize production issues, then projects, then enhancements. Enhancements can be categorized in terms of legal requirements and financial impact. So, if it is more finance or compliance-related, then definitely these are the things that we prioritize from an enhancement. Then all the others, like nice to have or has a work around can be delayed.

Perhaps the most essential role of direct managers is supervising the developers, not just for capacity planning but for their growth as well:

We try to track each resource's capacity to ensure that if the work is extensive, we do not let them take incident tickets. During Monday meetings, they relay that I am full this week, so I will not take additional tickets first. So, everyone needs to be transparent about their workload so we can support them. If they do not raise their hand, we will not know until they or something is being escalated.

There is a weekly meeting where we let the developers pick the area, especially if this is for a specific developer that wants to grow. Let us say there is a ticket not critical in the Finance area, and we have a month's timeline. We ask if they want to take it so that it is more of a learning experience for them as well. We are not pushing it to them unless it is tight. We try to give everyone the flexibility and freedom to make the changes they want.

As much as possible, we do not do it (reassign developments) unless there are times that someone is suddenly following up. We will check the impact because the developer is on leave. So, if there is not much impact or the business is just panicking, we wait until the person returns because it is hard to transfer developments, the line of thinking, and the train of thought.

Although we have coding standards, each developer will still have different ones.

During releases, the direct manager also has a hand in the deployment. Participant 2 shares his experience with waves or releases based on the number of customers:

In waves, instead of "Big Bang (full deployment), we go live by country. It is not a small release because it is complete, but considering all the requirements of, let us say, Asia Pacific. If it is 15 countries, then it will take a long time to align. So, one by one, let us adjust and learn from that.
Each country has different legal requirements and taxation laws, so there are also different ways of working. Some countries have manufacturing plants, and some don't. So, the handling would be different as we try to align everything.
As we started to globalize, we would have rolled out to multiple regions. So, we also need to coordinate and deploy with the different developers.

Change Management. Sometimes referred to as the CAB, they support and guide the business for project scheduling and deployment, especially during planning and releases.

They can directly influence the projects, like in Participant 3's case where the CAB is not Agile and slowing them down:

For us, the major roadblock for Agile is Change Management. With Agile, you implement more things faster and more frequently. However, the requirements are hefty because of our change management's waterfall approach. There are lots of documents and lots of change requirements that

must be filled out. Before, we only implemented once every few months, and now we implement every two weeks or so, which will require extensive resources on the change management side. From a development point of view, it is not that difficult because we are updating the documents and coding in parallel, so it gets to production faster.

For some releases, like in Participant 1's, the change management is purely a process for low-risk tasks. The board only gets involved when the development is high risk:

For smaller releases, it is more of a change management process that needs to be followed; it is more administrative than technical or functional. When specific changes are identified as routine, they do not go to CAB calls, primarily if they identify low-risk changes. So, the higher the risk, the more you must present it to technical people. If it is flagged as low risk, it usually does not require any technical work to be involved but more on following the change management process.

This participant also stated that the CAB is only concerned with the current changes you are presenting to them:

They do not have insight into what was previously implemented and what needs to be implemented. They usually are more concerned about what you are currently presenting. So, I am not sure if there are instances when they are made aware that this change will be needed for another one. Based on my experience, the change advisory board is just really concerned about your current change. So, they're not privy to or concerned if this is a prerequisite to some work changes.

Interactional:

Business Team. Not to be confused with the functional team, the business is an operational team that supports certain parts of the organization, like Sales or Marketing. They spearhead the projects themselves on behalf of the organization or the clients. They are the Customers in the old XP (Beck, 1999) and are responsible for the business processes. Their closest equivalent in the new XP team is the Users (Beck & Andres, 2004).

They usually work closely with the functional groups and interact less with the developers. As Participant 1 pointed out, they should be knowledgeable about the solution from a high-level perspective:

> *In addition to the usual users, they are aware of the technical settings and functionalities of what they're handling. I am not sure if we could tag them as technical people, but maybe we could consider them experts in the area.*

He adds that the business team, through the functional resources, provides test criteria that will not only affect their testing but also other tests as well:

> *In any development, the business also submits the test criteria. During planning and design, we agreed on the test approach and the technical changes that would need to be made. We also require the business or the project to submit their test scenarios. During development, we also do unit testing and need to know the areas in which we need to test. What are their expectations from them? That is why during the design, before the development, we request test scripts or scenarios that we will be doing for unit testing as well. Once we have the test scenarios, we do the development, and then we test based on the scenarios before we hand it over to them.*

For small releases in Participant 1's company, the business gets involved instead of the functional resources:

> *Small releases, like bug fixes, usually don't involve the functional resources anymore. You are more directly connected with the business that requested or wanted the improvement. So, these are minor changes that they're more familiar with. Usually, the functional resources test the changes, but for matured or already established solutions, it's the business that does the testing.*

The business also decides on the scope of the releases, but with the team's support, they do not have an idea of how IT works end-to-end, according to Participant 2. They get involved with the UAT and support the functional resources.

Interactional-Transactional:

External Developers. Unlike internal developers, external ones work outside the project team. In Participant 2's organization, they work with other development teams for inter-system solutions as early as planning:

> *There are times that there will be dependencies when working with other teams, like middleware or cross-box solutions. Let us say the X team has changes from their side, and we have changes on ours. So, we are trying to manage it because the X system is managed by a different team right now. We try to collaborate, coordinate our changes, move them simultaneously, or maybe work with other developers. So, there needs to be coordination between the back and front end.*

External developers can also be indirectly involved, especially when they are experts at specific platforms or technologies, like in Participant 3's situation:

There are cases when we could benefit from talking with other developers, like best practices or which technology to use for this requirement. Typically, the user wants this kind of output when there is a requirement. Technically, you have many choices and options for displaying that output. Sometimes, there are updates in the platform that we are working with to show that there is a newer or better way of doing things. This is when collaboration with other developers comes in because often, other developers will have more up-to-date knowledge of those kinds of things.

Participant 1 shares his experience as an external developer when a new programmer engages in the project he originally worked on:

It happens when the project ends, and they want to implement other solutions. Sometimes, I get involved because I was the original developer, so I need to explain to the new developer what was implemented: this was the approach, the requirements, and what we did in this part of development. We explain so that they get onboard a bit more quickly with what was implemented.

We usually present functional and technical specifications documents because sometimes you tend to forget what was implemented. We also usually refer to technical documents and sometimes handover documents that are presented to the business or to the project. So yeah, we usually refer to those kinds of documents to explain to other developers what's happening.

The expectation is that they know the code and just need to understand the flow. That is why it is not discussed at the code level; it is more about the details of how it works. So, maybe the objects and tables that you used, but not the exact coding.

Technical Teams. These are other technology groups that the developers depend on for their projects. They can be internal teams within the company or external third-party entities like vendors or consultants that provide infrastructure or services. In the old XP, they could be equivalent to Consultants or Programmers themselves (Beck, 1999). The Architect is the closest team member in the new XP (Beck & Andres, 2004).

Participant 1 works with various teams to assist him from the planning phase up to development:

> *During planning, we must know which systems you will use. So, prior to that, you usually ask the security folks assigned to provide you with access to those systems during the planning phase. You also coordinate with those who handle connectivity between systems to provide. So, at a high level, you will request those things during the planning phases. Sometimes, those handling master data are involved in the initial testing of your program. Afterward, the (conversation with the) technical usually follows when your development starts.*

Together with the developers, these technical teams are like puzzle pieces that complete the overall project. Beck's new XP version (2004) describes an overall overseer who is also a part of the technical team (sometimes the lead), the Architect, who keeps the big picture in mind as these groups focus on the smaller sections. The Architect is not mentioned in Beck's old version (1999), but he speaks of an equivalent consultant with deep technical knowledge.

Functional Resources. The members of this team are responsible for communicating with the clients for business requirements and providing solutions based on their knowledge of the systems. They act as the intermediary between the

developers and the clients, which can be customers or end users. In the old XP (Beck, 1999), they have no direct equivalent unless the Consultant has functional knowledge. In the new XP team (Beck & Andres, 2004), they can be Interaction Designers and Technical Writers.

Participant 2 describes them as such, which is crucial for the planning stage:

> *They are business process experts from the functional side. We are trying to do this (converse with them) because we do not want to directly communicate with the business because there will be a miss or a gap in translation. So, we want them to go through the business, align with the functional team, and then go to us (developers).*

It is the functional resources that start the ball rolling during planning. According to Participant 1:

> *Usually, the functional (resources) drive the discussion during the planning stage because they know the solution. They also know the limitations. That is why they are saying, okay, we will require some developments in this area. Then, sometimes, the developers are involved in the feasibility or the difficulty level. So, during the Planning Game, they usually ask the developer about the difficulty and how long it will take, but they do not ask for too much detail about it. They know that development will be needed and are more concerned about how long it can be done because of the timeline.*

Documentation is a must for functional resources. For Participant 2, if the functional team understands the processes, they will be able to document the specifications properly, and this will already be enough for the developers. As per Participant 1, it is not a technical document as it does not contain the exact coding

itself, but the logic is there; the high-level idea or the gist of what will be needed is documented.

There are challenges in communicating with functional resources when they assume that the developments are easy. Participant 1 narrates:

> *They (functional) always think that it is something that you could just do, like turning on a switch, but they really don't know sometimes or do not comprehend the development. Although it sounds simple, it is difficult to implement.*
>
> *For example, we want red to become blue. It sounds easy, but programming is done at the backend, and it's not that straightforward. Usually, that is due to a misconception or miscommunication between the functional (and the developers).*

For scenarios like this, the developers should communicate with the functional resources to dispute them (if needed) and offer suggestions. Participant 2 checks other solutions that might be more efficient, so they try to share them. Going back to Participant 1, if you have implemented a solution before, you are more confident in raising the issue that it is not that straightforward. You will need to explain some steps so that they understand where you are coming from. At the end of the planning stage, the functional resources will come up with the final direction.

During TDD, the functional resources come up with the test cases before the actual developments. Participant 2 elaborates:

> *We already have test scripts and test cases, and even our intake form for development has a section highlighted as test cases. So, when they raised the request to us, they also put some of the test cases there. It is already written, so we can understand how to test it out after the development.*

Any additional changes in the test cases will affect the timeline, especially the release. Participant 2 reveals that the timeline frequently moves because of never-ending requirements, like when specific business processes were missed, so it keeps expanding. Although the functional resources will inform the business regarding the delays, they will not necessarily take the blame:

> *As long as we explain clearly that the complications came from this and then the business would understand those processes, then it should be okay. They (functionals) base their functional specs and test cases on the business change requirement document. It is not really to blame who made the error at the end of the day if the business is unaware of its impact on the business process. The functional team will not consider it, and the developers will not. So, it is more important that everyone takes accountability and understands the impact and adjustments of time.*

Transactional-Interactional:

Internal Developers. The Programmer is mentioned as a role in the old XP roles (Beck, 1999) and is still included in the XP team in the new version (Beck & Andres, 2004). For this research, internal developers are a team of programmers working together on a project or product.

During planning, the developers should be prepared to handle specific areas. For Participant 2, it is one area per developer and having backups to lessen risks. The developers should be able to handle challenges that come their way, like what Participant 3 relayed:

> *When we want to do something or when the business wants something but we're not sure if it's technically feasible, we need to coordinate with*

developers and get their ideas. Then, we usually do a very rough prototype, maybe just a proof of concept (POC). This is to see if the POC is workable, even if it is very rough and many things are hard coded, as long as the main principle is being tested. Then, at least, we'll have a more definite answer if we can really commit to that change or reject it because we say it is not workable at this time.

Some companies have coding standards that developers should follow to guide them with the naming conventions, a form of communication through code. This practice was listed previously as one of the primary ones of XP (Beck, 1999) but was removed in the new version (Beck & Andres, 2004). Participant 1 emphasizes the importance of these rules even if the implementation is not usually followed:

If there are no strict standards, you follow your own. So, what are the best practices? You follow that.

Participant 3 has a different opinion on standards, that it is not necessary as long as it is readable by other developers:

Coding standards would be a good starting point, but it is unnecessary. I have seen our base coding standard, but the coding and even the coding standard itself have changed over the years. So, you see through the years that the different programs have different coding standards. In a way, it is not standardized; they all look different anyway. If the comments are complete, I do not care what exact coding standards you put in. It is okay with me as long as the variables you use make sense of what the logic does. So, I do not consider it the highest priority, like a nice to have, as it is not essential for my book.

Refactoring, or optimizing code occasionally, is another critical developer practice that was once listed as a practice (Beck, 1999) but was delisted (Beck & Andres, 2004). Some developers like Participant 3 still do this but only as needed:

> *We do it mostly when it is needed. When an actual incident is caught because of that program being too slow, performance issues are experienced. That is when we look at the code and optimize it, in that we would do all those performance analyses. We do not normally do this to clean up everything because there is not enough capacity to look at them. So, primarily, it is based on priority. If it is causing a problem, we will look at it. Many of our tools are like that; they are not the most optimized, but they are working and not really that slow, so it is okay.*

During releases, the developers must adhere to specific processes to push their developments to production. For Participant 1, this is where the developers communicate with the Change Management team:

> *You need to follow a change management process for these small releases and some approvals. As a developer, I will make the changes and then release or implement them in the production system. It is mostly administrative at this level.*

Discussion

For the selected developer practices, the only code-related result was TDD. Surprisingly, coding standards and code refactoring were not rated high enough to make it to the top three. These two highly involve the developers, but the participants did not consider their importance enough to merit high scores. The researcher agrees with the results since the dropped practices only benefit the developers and do not elicit enough communication.

Developer communication is essential for all practices. In the Planning Game, discussions between the developers and functional resources are necessary for the development's success. In TDD, testing is as crucial as development; communication is the key to avoiding rework or "surprises." Finally, communication is crucial for the success of Small Releases, especially for daily and weekly releases, where the risks are higher.

The TDD practice has most of the developer roles, with the Planning Game coming in a very close second. The developer should focus on his developments and tests during the TDD phase, but this is only sometimes true. All the necessary information should be addressed in the Planning Game to lessen the need for unnecessary communication in TDD. Any knowledge gaps will have a cascading effect on the Small Releases segment.

The Documenter role takes center stage, an essential for all practices. However, written communication is not the only essential type, as all roles involve verbal and visual communication as well.

As for the developer interactions, the Planning Game practice almost involves all the teams. This shows that ideally, communication with the different groups should start before the developments. Even though the Business Team is excluded,

they are still represented by Functional Resources, so basically, everyone is covered.

As this is developer communication, it is unsurprising that the Internal Developers are in the middle of the interactions. According to function, they are surrounded by three groups: direction (managers), support (functional and technical teams), and operations (business).

Chapter V

SUMMARY, CONCLUSION, AND RECOMMENDATIONS

Summary

In the first chapter, a brief background of the research is introduced. The research problems are presented together with the objectives and significance of the study, as well as its scope and limitations.

In the second part, a literature review is provided to help build up the research. As advised by Charmaz (2006), it is necessary to delay the completion of the literature review to let the insights emerge from the data. Initially, there is minimal information about SDLC, Agile, and XP, but additional information, such as CMC theories and Models of Communication, is added after data analysis.

In chapter three, the research design is introduced, with CGT as the research method to aid in deriving a theory and model. The participants and data collection and analysis based on the selected research instruments have been identified. The themes and codes used for analysis have also been presented.

The fourth one discusses the answers to the research objectives using questionnaires and interviews. The following XP developer communication concepts have come out: three XP practices that involve developers, 14 roles of the developers, and seven teams that interact with the developers grouped into three modes of communication. The results are also discussed.

Finally, this section provides a summary, a conclusion, and recommendations for further research.

Conclusion

This dissertation attempts to answer four research questions related to developer communication in XP practices:

1. Which XP practices involve the developers the most?
2. What are the different ways developers communicate in XP practices?
3. How do developers communicate in XP practices?
4. What model can be constructed for developer communication in XP practices?

For the first question, selected developers have been asked to rate XP practices according to their perceived order of importance. The top three emerging practices are Planning Game, TDD, and Small Releases. Their equivalent new practices are Stories, Weekly and Quarterly Cycles, and TFP.

As for the second question, coding has materialized 14 roles from the interviews classified into seven groups. They are as follows:

- Planning Game only: Negotiator and Researcher
- TDD only: Collaborator, Developer, and Tester
- Small Releases only: Interpreter
- Both Planning Game and TDD: Coordinator, Designer, Learner, Mentor, and Translator
- Both Planning Game and Small Releases: Assessor
- Both TDD and Small Releases: Presenter
- All three: Documenter

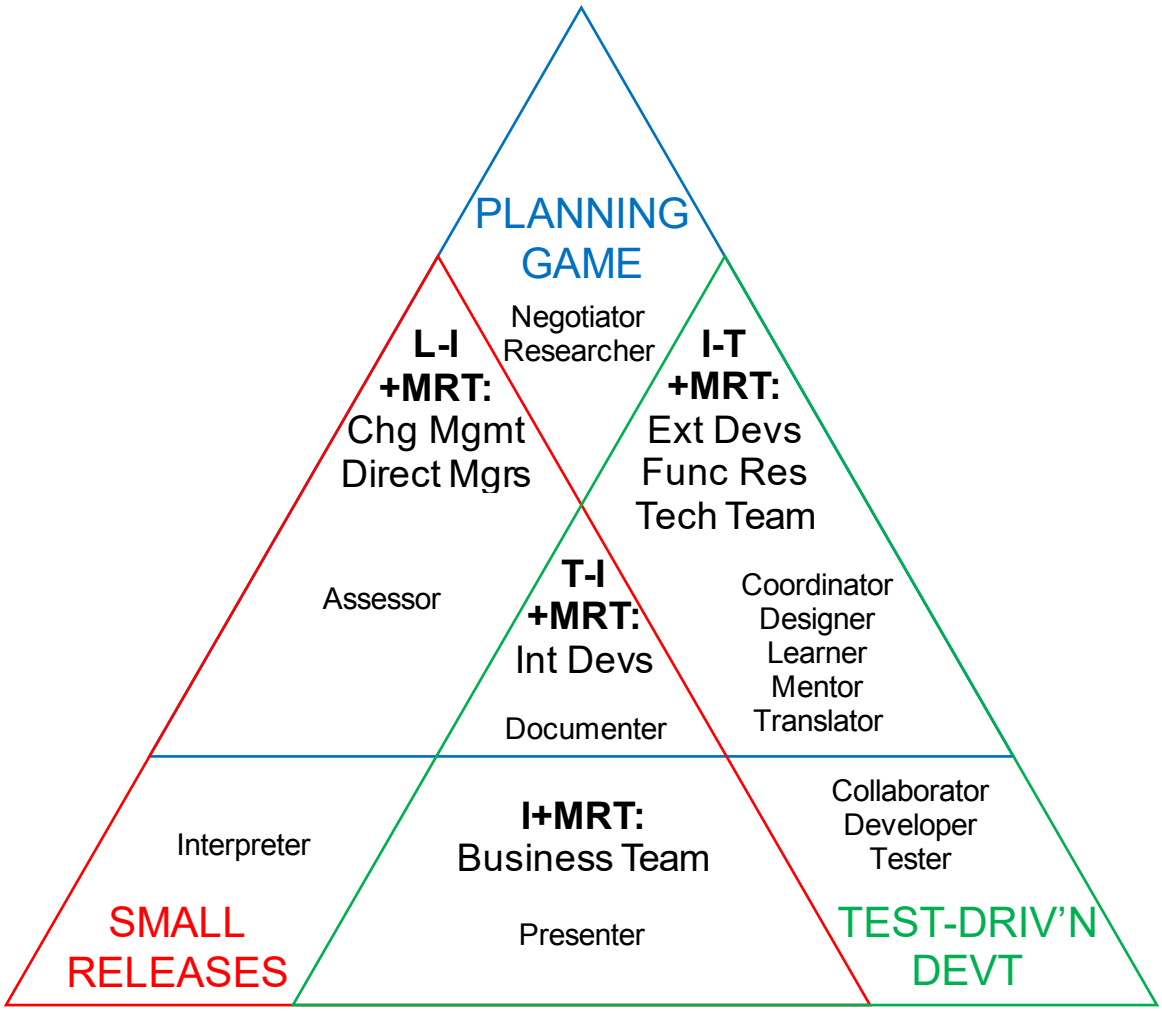
For the third question, it is revealed in the interviews that seven teams interact with the developers classified into four groups according to modes of communication:

- Linear-Interactional (L-I): Direct Managers and Change Management
- Interactional (I): Business Team
- Interactional-Transactional (I-T): External Developers, Functional Resources, and Technical Team
- Transactional-Interactional (T-I): Internal Developers

By combining Figures 4-3 and 4-4 from the previous chapter, an initial model of XPDC with all the practices, teams, and interactions present has been formed. Additionally, MRT has been added to all the teams as the literature mentions that communication is most effective with high media richness technologies.

With all these concepts combined, Figure 5-1 addresses the fourth question. According to Beck (1999), the developers are the heart of XP. Similarly, the internal developers are the core of our proposed XPDC triangle.

Figure 5-1: *XPDC Triangle*



The first component does not include all XP practices, but the selected three have all the components of SDLC – planning, analysis, design, and implementation. As for the second component, identifying different developer roles expands the new XP version; it states that these should not be fixed and rigid. For the third and final component, the Internal Developers are at the center of the different teams interacting with them. All three categories seem to cover XP's essential developer communication practices.

Recommendations

Here are some recommendations for further research on this topic:

1. Gather more data by conducting surveys with a representative sample to represent a larger population. This results in more interviewees with different classifications, such as age groups, experience, and programmer types. Aside from the traditional and end-user programmers, conversational programmers who seek programming skills not to write code but to communicate better with developers (Chilana, Singh, & Guo, 2016) can be added.
2. Come up with a more comprehensive set of questions for the interviews. Specific queries can be made since the practices, roles, and interactions have been identified.
3. Investigate the possibility of transactional communication through computer code among developers. By applying coding standards and placing code comments, it will be interesting to examine how code is used as a conversation medium and determine its degree of media richness.
4. Expand the coverage of the theory/triangle by initially including code-centric practices such as Continuous Integration and Pair Programming. Afterward, the other primary and corollary practices can be included.

BIBLIOGRAPHY

Ambler, S.W. (2002). *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process.* New York, USA: John Wiley & Sons, Inc.

Ambler, S.W. (2014). *Choose the Best Communication Technique Available*. Disciplined Agile. https://disciplinedagiledelivery.com/choose-the-best-communication-technique-available/.

Ambler, S.W. & Vizdos, M. (2008). *Agile Practices and Principles Survey 2008*. Ambysoft, Inc. http://www.ambysoft.com/surveys/practicesPrinciples2008.html.

Barnlund, D.C. (1970). *A Transactional Model of Communication*. In K.K. Sereno & C.D. Mortensen (Eds.), *Foundations of Communication Theory* (pp. 83–102). New York, USA: Harper.

Beck, K. (1999). Ex*treme Programming Explained: Embrace Change, 1st ed*. Boston, USA: Addison-Wesley.

Beck, K. (2002). *Test Driven Development: By Example, 1st ed*. Boston, USA: Addison-Wesley.

Beck, K. & Andres, C. (2004). *Extreme Programming Explained: Embrace Change, 2nd ed*. Boston, USA: Addison-Wesley.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marrick, B., Martin, R., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for Agile Software Development*. Agile Manifesto. http://agilemanifesto.org/.

Berlo, D.K. (1960). *The Process of Communication*. New York, USA: Holt, Rinehart, and Winston.

Blumler, J.G. & Katz, E. (1974). *The Uses of Mass Communications: Current Perspectives on Gratifications Research*. Beverly Hills, CA, USA: Sage.

Charmaz, K. (2006). *Constructing Grounded Theory: A Practical Guide through Qualitative Analysis*. London, UK: Sage Publications Ltd.

Chilana, P.K., Singh, R., & Guo, P.J. (2016, May 7). Understanding Conversational Programmers: A Perspective from the Software Industry. *ACM CHI '16: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 1462–1472. https://doi.org/10.1145/2858036.2858323.

Cockburn, A. (2002). *Agile Software Development*. Addison-Wesley Professional.

Copeland, L. (2001, December 3). *Extreme Programming*. Computerworld. https://www.computerworld.com/article/2585634/extreme-programming.html.

Corbin, J. & Strauss, J. (2014). T*he Basics of Qualitative Research: Techniques and Procedures for Developing Ground Theory, 4th ed*. CA, USA: Sage Publications Ltd.

D'Angelo, S. & Begel, A. (2017 May 2). Improving Communication Between Pair Programmers Using Shared Gaze Awareness. *ACM CHI '17: Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 6245–6290. https://doi.org/10.1145/3025453.3025573.

Daft, R.L., & Lengel, R.H. (1983, May). Information Richness: A New Approach to Managerial Behavior and Organization Design. *Report no. TR-ONR-DG-02*. Office of Naval Research.

Dennis, A., Wixom, B.H., & Roth, R.M. (2012). *Systems Analysis & Design, 5th ed.* New York, USA: John Wiley & Sons, Inc.

Drew, C. (2023, March 18). *Transactional Model of Communication: Examples & Definition*. Helpful Professor. https://helpfulprofessor.com/transactional-model-of-communication/.

Glaser, B.G., & Strauss, A.L. (1967). *The Discovery of Grounded Theory: Strategies for Qualitative Research*. New Jersey, USA: AldineTransaction, A Division of Transaction Publishers, Rutgers—The State University.

Kappelman, L., Jones, M.C., Johnson, V., McLean, E.R., & Boonme, K. (2016, July 22). Skills for Success at Different Stages of an IT Professional's Career. *Communications of the ACM, 59*(8), 64–70. https://doi.org/10.1145/2888391.

Korkola, M., Abrahamsson, P., & Kyllonen, P. (2006). A Case Study on the Impact of Customer Communication on Defects in Agile Software Development. *IEEE AGILE 2006 Conference (AGILE'06)*. http://dx.doi.org/10.1109/AGILE.2006.1.

Layman, L., Williams, L., Damian, D., & Bures, H. (2006 September). Essential Communication Practices for Extreme Programming in a Global Software Development Team. *Information and Software Technology, 48*(9), 781–794. ScienceDirect. https://www.sciencedirect.com/science/article/pii/S0950584906000024.

Kumar, S. (2016). *Communication Patterns and Strategies in Software Development Communities of Practice*. Doctoral dissertation, Michigan Technological University. https://doi.org/10.37099/mtu.dc.etdr/186.

Librero, F.R. (2012). *Writing your Thesis (A Practical Guide for Students)*. Los Baños, Laguna: University of the Philippines Open University.

Martin, A.M. (2009). *The Role of Customers in Extreme Programming Projects*. Doctoral thesis, Te Herenga Waka—Victoria University of Wellington.

https://openaccess.wgtn.ac.nz/articles/thesis/The_Role_of_Customers_in_Extreme_Programming_Projects/16959169.

Meier, J.D. (2019). *The 4 Circles of Extreme Programming*. JD Meier. http://jdmeier.com/4-circles-of-extreme-programming.

Papacharissi, Z. & Rubin, A.M. (2000, June 7). Predictors of Internet Use. *Journal of Broadcasting & Electronic Media, 44*(2), 175–196. https://doi.org/10.1207/s15506878jobem4402_2.

Pinho, D. & Aguiar, A. (2020). The AgilECo Pattern Language: Physical Environment. *EuroPLoP '20: Proceedings of the European Conference on Pattern Languages of Programs 2020*, art. 30, 1–9. https://doi.org/10.1145/3424771.3424790.

Salancik, G.R. & Pfeffer, J. (1978). A Social Information Processing Approach to Job Attitudes and Task Design. *Administrative Science Quarterly. 23*(2), 224–253. https://doi.org/10.2307/2392563.

Schramm, W. (1954). *How Communication Works*. In W. Schramm (Ed.), *The Process and Effects of Communication* (pp. 3–26). Urbana: University of Illinois Press.

Shah, S.M. & Amin, M. (2013). *Investigating the Suitability of Extreme Programming for Global Software Development*. Master's thesis, Bleckinge Institute of Technology. Digitala Vetenskapliga Arkivet.

Shannon, C. & Weaver, W. (1949). *The Mathematical Theory of Communication*. Urbana: University of Illinois Press.

Short, J.A., Williams, E., & Christie, B. (1976). *The Social Psychology of Telecommunications*. Wiley.

Srinivasan, R.M., Jetcheva, J.G., & Chander, A. (2017, August 24). Last Mile End-User Programmers: Programming Exposure, Influences, and Preferences of the Masses.

2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), 262–264. https://doi.org/10.1109/ICSE-C.2017.129.

Thomas, D. & Hunt, A. (2019). *The Pragmatic Programmer: Your Journey to Mastery, 20th anniversary edition*. Boston, USA: Addison-Wesley Professional.

Urai, T., Umezawa, T., & Osawa, N. (2015, June 22). Enhancements to Support Functions of Distributed Pair Programming Based on Action Analysis. *ITiCSE '15: Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, 177–182. https://doi.org/10.1145/2729094.2742616.

Walther, J.B. (1992). Interpersonal Effects in Computer-Mediated Interaction: A Relational Perspective. *Communication Research, 19*(1), 52–90. https://doi.org/10.1177/009365092019001003.

Wells, D. (2013). *Extreme Programming: A Gentle Introduction*. Extreme Programming. http://www.extremeprogramming.org/index.html.

Wood, A.F. & Smith, M.J. (2005). *Online Communication: Linking Technology, Identity, & Culture (2nd ed.)*. Routledge.

Wrench, J.S. & Punyanunt-Carter, N.M. (2007). The relationship between computer-mediated-communication competence, apprehension, self-efficacy, perceived confidence, and social presence. *Southern Journal of Communication, 72*(4), 355–378. https://doi.org/10.1080/10417940701667696.

Xiaohu, Y., Bin, X., Zhijun, H., & Maddineni, S.R. (2004). Extreme Programming in Global Software Development. *2004 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2004-CCGEI 2004)*, 1845–1848. https://www.cin.ufpe.br/~cct/sbes2008/Xiaohu2004.pdf.

Yermolaieva, S. (2020, December 21). Communication Challenges in Agile Teams from the Communication Theory Prospective. *ESSE '20: Proceedings of the 2020 European Symposium on Software Engineering*, 88–95. https://doi.org/10.1145/3393822.3432327.

Zarb, M., Hughes, J., & Richards, J. (2015, February 24). Further Evaluations of Industry-Inspired Pair Programming Communication Guidelines with Undergraduate Students. *ACM Special Interest Group on Computer Science Education (SIGCSE) 2015:* Proceedings *of the 46th ACM Technical Symposium on Computer Science Education*, 314–319. https://doi.org/10.1145/2676723.2677241.

Zieris, F. & Prechelt, L. (2020, October 1). Explaining Pair Programming Session Dynamics from Knowledge Gaps. *ICSE '20: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 421–432. https://doi.org/10.1145/3377811.3380925.

Appendices

APPENDIX A

Questionnaire on Software Development Practices

A. About Yourself

1. Please enter your full name; this will constitute your electronic signature for agreeing to participate.
2. What are your current and past IT roles/responsibilities? Check all that apply.
    i. Developer/Programmer
    ii. Systems/Business Analyst
    iii. Functional/Technical Consultant
    iv. Support
    v. UI/UX Designer
    vi. Tester
    vii. Quality Assurance
    viii. Technical/Team Lead
    ix. Project Manager
    x. Product Owner
    xi. End User
    xii. Client/Customer
    xiii. Other
3. Do you consider yourself as one of these types of developers?
    i. Traditional Developer - creates programs with established programming skills
    ii. End-user Developer - creates programs with no/basic programming skills

iii. Conversational Developer - creates programs to communicate better with traditional programmers
iv. I don't consider myself as a Developer

B. Software Development Life Cycle (SDLC)

1. Which SDLC methodologies have you used at work? Check all that apply.
    i. Waterfall (e.g., Linear, Parallel, V-Model)
    ii. Rapid Application Development or RAD (e.g., Iterative, Prototyping)
    iii. Agile (e.g., Scrum, Kanban, Lean, Extreme Programming)
    iv. Other
2. Which SDLC methodology have you used the most?
    i. Waterfall
    ii. RAD
    iii. Agile
    iv. Other
3. Which Agile frameworks have you used at work? Check all that apply.
    i. Scrum
    ii. Kanban
    iii. Lean
    iv. Bimodal
    v. Hybrid
    vi. Crystal
    vii. Extreme Programming (XP)

viii. None/Not applicable

ix. Other

C. Extreme Programming

1. Which XP practices have you used at work? Check all that apply.

i. On-site Customer - Client sits with the team to resolve issues, sets priorities and scope, and provides test scenarios.

ii. Planning Game - Client decides on the scope based on developer estimates. Developers only implement the functionalities approved by the client.

iii. Sustainable Pace (40 hour week) - No one should work a second consecutive week of overtime.

iv. System Metaphor - System is defined by a metaphor defined by the client and developers.

v. Simple Design - The design runs all the tests, says everything the developers want to communicate, and has efficient code.

vi. Small Releases - New releases are made often instead of one big release.

vii. Coding Standards - Developers choose variables/names in the same style and code is formatted in the same way.

viii. Collective Code Ownership - Any developer improves the shared code anytime they see fit.

ix. Continuous Integration - New/Changed code are integrated in the system within a few hours. The code must be fixed for any unsuccessful tests, otherwise the changes must be discarded.

x. Test-Driven Development - Developers conduct unit tests; clients write functional tests. All tests should run correctly.

xi. Code Refactoring - Optimize existing code without compromising the system to make it efficient, including the removal of duplicate/unnecessary code.

xii. Pair Programming - Code is written by two people at once using a single shared computer set.

xiii. None/Not applicable

2. What is your opinion on XP practices?

APPENDIX B

Questionnaire on XP Practices

Previously, I have asked which XP practices you have used at work. I have listed the top five answers in no particular order. Wearing your traditional/end-user developer hat, please rate each item in your perceived order of importance. We are using the Likert scale of 1 to 5, where 1 is the lowest and 5 the highest rating.

Your answers will be collated with the others, and only the top two or three items will be considered for the interview.

1. Planning Game - Client decides on the scope based on developer estimates. Developers only implement the functionalities approved by the client.
2. Small Releases - New releases are made often instead of one big release.
3. Coding Standards - Developers choose variables/names in the same style and code is formatted in the same way.
4. Test-Driven Development - Developers conduct unit tests; clients write functional tests. All tests should run correctly.
5. Code Refactoring - Optimize existing code without compromising the system to make it efficient, including the removal of duplicate/unnecessary code.

APPENDIX C

Interview Questions on XP Practices

Description:

Semi-structured interview on the resulting top three XP practices in order of importance: Planning Game, Small Releases, and Test-Driven Development.

A. Pre-Questions
    1. How long have you been working in the IT industry? As a developer?
    2. Do you agree with the ranking of the XP practices? If not, would you change your ranking?

B. Guide Questions
    1. In what role/capacity did you use these XP practices? Agile or not?
    2. Based on experience, what are the communication strengths and weaknesses of each? Challenges?
    3. During your stint as a traditional/end-user developer, how did you communicate these practices
        i. with other developers?
        ii. with non-developers?