Chapter 1
Introduction

When defects in an existing paradigm accumulate to the extent that the paradigm is no longer tenable, the paradigm is challenged and replaced by a new way of looking at the world.
Dawn Freshwater and Gary Rolfe

Bauer coined the term "software engineering" (SE) forty years ago at a NATO Software Conference, expressing a need: "What we need is software engineering". In the last decade, an incredible impact on research and practice in SE has been made by the agile movement. Under this broad umbrella of the agile methodologies sit specific approaches such as eXtreme Programming (XP) [23, 25, 129, 247], Scrum [220, 221], Lean Software Development [206], etc. They share similar values and beliefs, which have been documented in the "agile manifesto" [24]. Among the agile methodologies, XP is probably the most prominent one and an empirical evaluation of one of its key practices, called Test-First Programming (TF), is a general aim of this book.

1.1 Test-First Programming

According to DeMarco, "XP is the most important movement in our field today." [25]. This statement emphasizes the importance of XP as a new software development methodology.

XP can be seen from various perspectives, such as: a mechanism for social change, a software development methodology, a constant path to perfection, and an attempt to bring together humanity and productivity in software development [248]. XP is founded on five abstract but universal values (communication, simplicity, feedback, courage, respect) and tangible practices (e.g. test-first programming, pair programming) that are bridged together by certain principles (e.g. mutual benefit) [23]. According to Beck [23], values are the large-scale criteria we use to judge what we see, what we think and what we do; values also underlie our immediate and intuitive recognition of what we accept and what we reject in a given situation. Making values explicit is important, as, without values, practices (which are extremely situated) lose their purpose and direction. However, there is a gap between values and practices, since values are too abstract to directly guide development. Therefore,

L. Madeyski, *Test-Driven Development*,
DOI 10.1007/978-3-642-04288-1_1, © Springer-Verlag Berlin Heidelberg 2010

principles act as a bridge between values and practices. A detailed description of XP, its values, principles and practices is given by Beck [23].

This book focuses on one agile software development practice promoted by XP, i.e. Test-First Programming (TF) [23], also known as Test-Driven Development (TDD).[1] However, the impact of another XP practice, called pair programming (PP), is also taken into account in the first experiment to uncover possible interaction between both XP practices. TF and PP are considered not only the flagship and the most influential practices of XP methodology [15, 23, 144] but also the most controversial ones [179]. Classic development techniques, to which TF and PP are often compared, are Test-Last Programming (TL), also known as Test-Last Development (TLD) and Solo Programming (SP).

The key characteristic of TF is that programmers write tests before related pieces of the production code (i.e. before they change the behaviour of the production code, they must have a failing test) [22, 23, 72, 179]. The main characteristic of PP is that two programmers work on the same task using one computer and one keyboard [14, 23, 252, 254]. Both practices are presented in detail in Sects. 3.3.1.1 and 3.3.1.2 along with graphical process models that allows developers to apply the techniques. Of all of the practices of XP, TF is perhaps the most counterintuitive [179]. On the other hand, TF is, according to McBreen [179], the most powerful of XP practices for promoting the necessary paradigm shift for understanding and benefiting from XP.

TF has gained recent attention in professional settings [15, 22, 23, 97, 144, 195] and has made initial inroads into software engineering education [66, 67]. A wide range of empirical studies on the impact of both the TF and the PP practice in industrial [14, 28, 40, 56, 57, 87, 88, 126, 174, 192, 196, 217, 254, 255, 268] and academic environments [72, 79, 89, 91, 94, 111, 126, 157–161, 185, 187, 193, 194, 201, 252, 254, 257] give compelling evidence of their popularity.

1.1.1 Mechanisms Behind Test-First Programming that Motivate Research

Literature presents several interesting mechanisms behind TF adoption. The search for the possible consequences of those mechanisms underlies the empirical investigation carried out and expanded in this book.

TF provides instant feedback as to whether new functionality has been implemented as intended and whether it interferes with previously implemented functionality [72]. TF encourages developers to dissect the problem into small, manageable programming tasks to maintain focus and to provide steady, measurable progress [72]. Maintaining up-to-date tests gives courage to refactor [81] mercilessly in order to keep the design simple and to avoid needless clutter and

[1] According to Koskela [144], TDD and TF are just different names for the same practice and both may be used interchangeably.

complexity. Up-to-date and frequently run tests written for any piece of the production code that could possibly break [129] help to ensure a certain level of quality [72] and an acceptable level of test coverage as a side effect.[2] Moreover, tests provide the context for the making of low-level design decisions (concerning how classes and methods are named, what interfaces they provide and, consequently, how they are used) [72]. Tests are also perceived as another form of communication and documentation. Since unit tests exercise classes and methods, the source code of the tests becomes the critical part of system documentation. Unit tests communicate the design of a class because they show concrete examples of how to exercise the class's functionality [36]. A noteworthy aspect of TF is that it addresses the fears about staff turnover in a complementary way to the PP practice. The suite of unit tests is a safety net and repository of design decisions, so even if a new team member makes a coding mistake, it is highly likely that the suite of unit tests will detect the error [179]. Therefore, TF supports the refactoring and maintenance activities. Without the safety net, the developers would be very reluctant to change the design of the existing code [179].

A similar set of influential mechanisms behind PP adoption can also be presented. Therefore, synergy between both the TF and the PP practice will be investigated as well. Some researchers argue, two distinct roles (i.e. the role of a driver and a navigator) may be recognized when using the PP practice [14, 161, 254, 258]. The limited ability to think simultaneously at both the strategic and the tactical level can be easily addressed by the aforementioned distinct roles focused on different thinking levels. Furthermore, two distinct roles contribute to a synergy of the individuals in the pair [14, 161] as "two heads are better than one". PP also turns out to be a way of coping with the risks associated with bringing new programmers into the team [179]. The problem of the limited spread of knowledge inside the team may be dealt with by rotating pairs frequently [153, 253], as active pairing requires every pair to talk about the design, the requirements, the tests and the code. Also, active pairing ensures compliance with the rest of the programming practices, because the entire team can see who does not really comply with the rules [179]. Furthermore, one can try to reduce the risk of a high-defect rate because pairing acts like a continuous code review [179, 183–185]. Last but not least, PP has an effect on how programmers work, since it requires that all production code is written with a partner [23].

The aforementioned mechanisms behind TF and PP may affect many aspects of the software development, which includes code quality (e.g. percentage of acceptance tests passed, design complexity measures) or development speed (e.g. the number of acceptance tests passed per development hour). On the other hand, sceptics argue that such approaches to programming would be counterproductive. Managers sometimes feel they get two people to simply do the same task, thus wasting valuable "resources". At the same time, some programmers are long conditioned to working alone and resist the transition from solo programming (SP) to

[2] Code coverage analysis is sometimes called test coverage analysis but both terms are synonymous [53] and will be used interchangeably.

the PP practice [258]. Therefore, convincing all the members of development teams (e.g. programmers, managers) to accept the pair programming work culture may be a difficult task. TF is also not without difficulty to start with. Developers sometimes resent having to change their habit and to follow strict rules of the TF practice [162] instead of the TL (test-last) practice or even "code and fix" chaotic programming [80]. Furthermore, continuous testing and refactoring does not extend functionality. Better code or tests quality, as well as higher development speed, might be good counter-arguments in such situations. Therefore, this research objective is to empirically evaluate the hypothesized effects of the TF practice.

After this brief introduction to the agile movement in general and TF, PP as well as XP in particular, the rest of this chapter is structured as follows: general introduction to empirical research methodology is given in Sect. 1.2; the fundamentals of software measurement with respect to software quality and software development productivity are presented in Sect. 1.3; research questions are introduced in Sect. 1.4, while book organization and claimed contributions of the book are presented in Sects. 1.5 and 1.6, respectively.

It is worth mentioning that topics related to both research methodology and software measurement, if easily available and thoroughly addressed by existing books or research papers, have been dealt with briefly and corresponding references have been given, while more specific aspects of the conducted experiments and their analysis are covered in detail in other chapters.

1.2 Research Methodology

Research methodology presented in this section starts with a short introduction to the empirical software engineering movement in Sect. 1.2.1, while empirical methods are discussed in Sect. 1.2.2.

1.2.1 Empirical Software Engineering

The quality of the methods used to evaluate new methods, processes, technologies etc. indicates the maturity of the SE research discipline. Although the term SE was coined 40 years ago, the SE discipline is as yet not mature enough because there is still the need to make a transition from the software development based on presumptions, speculations and beliefs, to that based on facts and empirical evidence [132].

SE was formally defined by IEEE as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software" [113]. Consequently, experimentation, as a systematic, disciplined quantifiable and controlled way of evaluating new techniques, methods, practices, processes, technologies or tools [259], has become a fundamental part of research and practice in SE [227] and lays the foundation for this book.

Empirical software engineering (ESE) requires the scientific use of quantitative and qualitative data to understand and improve software development products and processes. ESE can be viewed as a series of actions to obtain evidence and a better understanding about some aspects of software development. An experiment can be performed to prove or disprove stated hypotheses. Software development practices, processes, technologies and tools have to be empirically evaluated in order to be better understood, wisely selected (among different alternatives) and deployed in appropriate contexts. Higher quality or productivity in SE would not be possible without well-understood and tested practices, processes etc. Recognizing that need, top level SE journal editors and conference organisers tend to expect empirical results to back up the claims or even select overtly empirical papers (e.g. *Empirical Software Engineering* journal by Springer, *International Symposium on Empirical Software Engineering and Measurement*, *International Conference on Software Engineering*, *International Conference on Product Focused Software Process Improvement*).

1.2.2 Empirical Methods

Empirical research methods in SE are used to explore, describe and explain phenomena. The empirical methods presented in this section encompass introduction to qualitative and quantitative research paradigms, fixed and flexible research designs, empirical strategies (e.g. experiments, case studies, surveys), as well as between-subjects and repeated measures experimental designs.

1.2.2.1 Qualitative and Quantitative Research Paradigms

A distinct feature that marks qualitative research is that it does not seek to produce quantified answers to posed research questions but can substantially contribute to the understanding of the subjects' perspective and a particular context within which the subjects (or participants) act (and the influence of that context), when developing explanations, generating theories etc. [175].

Quantitative research is often based on initial qualitative work (e.g. once research objectives are defined). That research consists in the systematic collection of data, which results in the quantification of relationships or characteristics of groups. Hence, it is particularly useful for finding quantitative answers and is well suited for testing hypotheses.

1.2.2.2 Fixed and Flexible Research Designs

Two main types of research designs are the fixed and the flexible design. The fixed research design typically involves a substantial amount of pre-specification about what should be done and how it should be done (e.g. the collection of quantitative data from two or more compared groups) [210]. The flexible research design, also

called qualitative, relies on qualitative data (in many cases in the form of words) and requires less pre-specification [210]. Mixed-method designs are also possible.

1.2.2.3 Empirical Strategies

The three most common empirical methods (or strategies) used in ESE are experiments, case studies and surveys. They have been discussed in detail by several researchers [181, 210, 267] and presented briefly in the forthcoming sections.

Experiments

A formal experiment is a controlled, rigorous investigation of an activity, where independent variables (IVs) are manipulated to document their effects on dependent variables (DVs). Formal experiments require a great deal of control, and therefore they tend to be small (so-called "research in the small"), i.e. involving small numbers of subjects or events [74].

Experiments are viewed by many as the "gold standard" for research [210] and are particularly useful for determining cause-and-effect relationships. Experiments provide a high level of control and are usually done in a laboratory environment [259]. In a true experiment, the experimenter has a complete control over the independent variable, e.g. the timing of the experimental manipulations, measurements of the dependent variable etc. Furthermore, in a true experiment there is a random allocation of subjects to the groups (two or more) and different groups get different treatments (e.g. development techniques). The objective is to manipulate one or more variables and to control the other variables that can influence the outcome (dependent variables). The effect of the experimental manipulation is measured and analysed by means of statistical techniques with the purpose of showing which technique is better.

Sometimes, especially in (close to) real-world situations, it is not feasible to conduct true experiments, and we have to accept a quasi-experimental design. In a quasi-experimental empirical study, the experimenter does not have complete control over manipulation of the independent variable as well as over the assignment of the subjects to the treatments. For example, it may be impossible to allocate the subjects randomly to the treatment groups (i.e. to different levels of the independent variable) for practical or ethical reasons. Although the same statistical tests can be applied, the conclusions from quasi-experimental studies cannot be drawn with as much confidence as from the studies employing true experimental designs (e.g. due to the lack of random allocation of the subjects to the treatments).

Single case experiments (labelled commonly as "small-N experiments" or "single subject experiments") focus on the effects of a series of experimental manipulations on the individuals who act as their own control. Experiments are prime examples of fixed research design [210].

Case Studies

A case study is, as suggested by its name, a study of a case (project, individual, group, organization, situation etc.), or a small number of related cases, taking its context into account [210, 267]. A case study is a research technique that makes it possible to identify the key factors that may affect the outcome of an activity, and to document the activities (their inputs, outputs, resources and constraints).

In case studies, one usually investigates a typical project (so-called "research in the typical"), rather than trying to capture information about a wide range of possible cases [74]. This research strategy involves the collection of qualitative data but may also include quantitative data [210]. Case studies are examples of flexible research design [210].

Surveys

A survey is an investigation in which data are collected from a population, or a sample from that population, through some form of interviews or questionnaires aimed at describing accurately the characteristics of that population (e.g. to describe the subjects' background, experience, preferences). The results from the survey are analysed in order to make generalisations, validate the effects of experimental manipulation etc. It is possible to poll over large groups of subjects or projects (so-called "research in the large") but it is not possible to manipulate variables as in case studies and experiments [74].

Surveys fall into two categories of research design: flexible and fixed, depending on degrees of pre-specification (e.g. open-ended interviews are usually flexible, while questionnaires with closed questions are typically fixed).

1.2.2.4 Between-Groups and Repeated Measures Experimental Designs

Between-groups (independent groups or between-subjects) experimental designs take advantage of separate groups of subjects for each of the treatments in the experiment and each subject is tested only once. There are some benefits from between-groups designs. Firstly, it is their simplicity. Secondly, their usefulness when it is impossible for a subject to participate in all treatments (e.g. when sex or age is one of the independent variables or when treatments alter the subjects irreversibly). Thirdly, the effect of the subjects' fatigue in course of the subsequent treatment occasions (e.g. when the subjects become bored or tired) is reduced, as the subjects participate in only one treatment occasion. However, there are also serious disadvantages of between-groups designs, mainly the following: a lower ability to detect cause–effect relationships (i.e. insensitivity to experimental manipulations) and expense in terms of the number of subjects, effort and time. Therefore, repeated measures (i.e. within-subjects) experimental designs are often preferred.

In repeated measures (within-subjects) experimental designs, each subject is exposed to all of the treatments in the experiment, so that two or more measures are collected for each subject. In consequence, the number of subjects, effort and

time are minimized, while sensitivity is increased due to the fact that variability in individual differences between the subjects is removed. In fact, each subject serves as his own control in the repeated measures design. On the other hand, treatments have to be reversible which is a kind of limitation. Moreover, carry-over effects from one condition to another have to be taken into account. The issues related to carry-over effects are discussed further in Sect. 4.7.3.

Mixed or hybrid experimental designs involve a combination of both of the aforesaid experimental designs.

1.3 Software Measurement

This section presents the fundamentals of software measurement, while more details can be found in [74].

Measurement is defined as "a mapping from the empirical world to the formal, relational world. Consequently, a measure is the number or symbol assigned to an entity by this mapping in order to characterize an attribute" [74]. This definition encompasses attributes of entities. An entity is an object (e.g. a developer, a software product) or an event (e.g. the maintenance phase of a project) in the real world, while an attribute is a property of an entity (e.g. active development time spent on the production code) [74].

Measurement helps us to understand the current situation and to establish baselines useful to set goals for the future behaviour. By measuring the inputs and the outputs of an object under study we are able to investigate and understand the effects of an experimental manipulation. Thus measurement is a crucial activity in empirical studies. Moreover, before we can expect to improve software processes and products, we must measure them. So measurement is important for understanding, control and improvement or corrective activities [74]. DeMarco emphasizes in his famous statement: "You can't control what you can't measure." [59].

Kan classifies software metrics into the following three categories: product metrics, process metrics and project metrics [135]. He also argues that software quality metrics are more closely associated with product and process metrics than with project metrics.

A short introduction to measurement levels is given in Sect. 1.3.1. The issues of software product quality and software development productivity are discussed in Sect. 1.3.2 and 1.3.3, respectively. Both sections are an introduction to research questions formulated in Sect. 1.4.

1.3.1 Measurement Levels

There are four levels at which variables can be measured: nominal (categorical), ordinal, interval and ratio levels. All that may be said about nominal data is that things with the same number are equivalent, while things with different numbers are

not equivalent [76]. Consequently, nominal data should not be used for arithmetic; however, it is possible to compute frequencies.

Ordinal data offer more information than nominal data, as the ordinal scale allows us to imply order or rank. For example, the best subject, who has a rank of 1, is better than the next best subject, who has a rank of 2. However, ordinal data tell us nothing about the relative differences, e.g. how better one of the subjects performed than the other one. Consequently, ordinal data need to be analysed with non-parametric tests. It is worth mentioning that a lot of data from questionnaires are ordinal.

Interval and ratio data are even more useful than ordinal data. For interval data, it is required that equal intervals between different points on the scale represent the same difference in the property being measured at all points along the scale. Ratio data have the same properties as interval data, but in addition, the ratios of values along the scale are meaningful. For example, the temperature scale in Celsius is an interval scale, since 2°C is not twice as hot as 1°C, in opposite to, for example, development time (something that lasts, say, 2 minutes is twice as long as something that lasts 1 minute). The difference between those two measurement levels is not critical. For example, the statistical package SPSS version 14.0 (SPSS Inc., Chicago, IL, USA), used for analysis, does not attempt to distinguish between them, and it employs, instead, the term "scale" to describe both measurement levels. Interval and ratio data can be analysed with parametric tests.

1.3.2 Software Product Quality

There is a wide range of views of what "software quality" is. A famous statement, attributed to Kitchenham (confirmed in personal communication from Barbara Kitchenham, July 2006), is that software quality is hard to define, impossible to measure and easy to recognize. Another famed statement concerning quality is that "quality, like beauty, is very much in the eyes of the beholder" [74].

Unfortunately, as for the measuring software quality, we have neither an accepted understanding of what quality is, nor commonly accepted software quality measures [130], even though a lot of quality standards and models have been proposed [80, 130]. The software quality standard ISO 9000-3 states that: "There are currently no universally accepted measures of software quality..." but "the supplier of software products should collect and act on quantitative measures of the quality of these software products."

1.3.2.1 ISO/IEC 9126

One of the most widely known software quality standards is the ISO/IEC 9126 standard. It is divided into four parts [121–124]. The first part [121] presents product *quality model*, explains the relationships between the different approaches to software quality and presents the quality characteristics and sub-characteristics that influence the quality of software products. The ISO/IEC 9126 standard provides different views of product quality which are closely related. According to the Quality

Model Framework Lifecycle of ISO/IEC 9126 [121], process quality influences internal quality which in turn influences external quality, which influences quality in use [30]. If experimental treatment (e.g. the TF practice) influences process quality, then it also affects *internal metrics*, *external metrics* and then *quality in use metrics*, as maintained by the aforementioned Quality Model Framework Lifecycle.

External Metrics of Product Quality

The second part of the standard [122] describes *external metrics* of product quality used to measure the characteristics (i.e. functionality, reliability, usability, efficiency, maintainability an portability) and related sub-characteristics (e.g. interoperability) identified in the aforementioned quality model. This external view of software quality focuses on the software dynamic aspect and is concerned with the completed software executing on the computer hardware, with real data [30]. The percentage of acceptance tests passed (PATP), called "reliability" by Müller and Hagner [187], can be considered an example of an *external metric* and is used in this book. *External metrics* of product quality can indicate the *quality in use* (e.g. if the percentage of acceptance tests passed is low, then expected end-user satisfaction and effectiveness is likely to be low). Research question related to the external view of software quality is posed in Sect. 1.4, while related research goal is formulated in Sect. 3.1. PATP is presented in Sect. 3.3.2.1, while the impact of TF on that *external metric* is analysed in Chap. 5.

Internal Metrics of Product Quality

The third part of the standard [123] presents the *internal metrics* used to measure the same collection of characteristics and sub-characteristics as in the second part of the standard [122]. However, the internal view of software quality concerns mainly static properties of the software product individual parts, including complexity and structure of the design and code elements. The advantage of *internal metrics* is that they can be used to measure quality properties in the early stages of development. Moreover, *internal quality metrics*, as opposed to *external quality metrics* as well as *quality in use metrics*, are meaningful on their own, i.e. do not depend on the hardware, the data etc. Class-level metrics proposed by Chidamber and Kemerer (hence labelled as CK metrics) [43] are, as mentioned by Bøegh [30], commonly used *internal metrics*. Higher level design quality metrics, proposed by Martin [171, 172], and the CK metrics may complement each other to spot weaknesses in the software architecture. *Internal metrics* can act as early indicators of external quality. Research question related to the internal view of software quality is posed in Sect. 1.4, while the related research goal is formulated in Sect. 3.1. Both internal quality metrics suites (i.e. the CK metrics and Martin's metrics) as well as the references to the underlying theory about the relationship between the OO metrics and some quality characteristics (e.g. maintainability) are presented in Sect. 3.3.2.2. However, it is worth mentioning that only some of the proposed *internal metrics* have been empirically confirmed as very significant for assessing

maintainability or fault proneness [19, 32, 33, 95, 200, 238]. The impact of TF on those *internal metrics* is analysed in Chap. 7.

Quality in Use Metrics

The fourth part [124] describes *quality in use metrics* and embraces the metrics used to measure the effects on the user, i.e. end-user productivity, effectiveness, satisfaction and safety. The *quality in use* view refers to the final product used in the real environment and conditions. That view is out of the scope of the empirical evaluation presented in this book, since the subjects used in the conducted experiments were developers, and not end-users, of the software products.

1.3.2.2 Test Code Metrics

Test code quality may be considered as a new perspective on software product quality. The quality of tests can indicate the quality of the related production code [201], especially when writing tests is a part of the development practice. Therefore, the goal is to shed light on the effects of the experimental manipulation from a different, test code, perspective. Consequently, research question related to this view of software product quality is posed in Sect. 1.4, while the related research goal is formulated in Sect. 3.1. The thoroughness and the fault detection effectiveness of unit tests are described in Sect. 3.3.2.4, while the impact of the TF practice on unit tests characteristics is analysed in Chap. 8.

1.3.2.3 Validity of Software Quality Standards

The ISO/IEC 14598 series [114–116, 118–120] defines a software product evaluation process, based on the ISO/IEC 9126. Both the ISO/IEC 9126 and the ISO/IEC 14598 series share a common terminology. The SQuaRE (Software Product Quality Requirements and Evaluation) project has been created to make them converge into the ISO/IEC 25000 series [125], as an attempt at eliminating the existing gaps, conflicts or ambiguities.

Several doubts concerning software quality standards have been raised by researchers. Pfleeger et al. [203] suggested that "standards have codified approaches whose effectiveness has not been rigorously and scientifically demonstrated. Rather, we have too often relied on anecdote, 'gut feeling', the opinions of experts, or even flawed research rather than on careful, rigorous software engineering experimentation." Al-Kilidar et al. [9] conducted a large experiment with 158 subjects and concluded that ISO/IEC 9126 is not suitable for measuring design quality of software products which casts serious doubts on the validity of the standard as a whole. Also, Kitchenham [139] argues that the selection of quality characteristics and sub-characteristics seems to be rather arbitrary and not clear (e.g. "it is not clear why portability is a top-level characteristic but interoperability is a sub-characteristic of functionality."). Also Arisholm [13] called the ISO/IEC 9126 quality model into question (e.g. "why is adaptability not a sub-characteristic of maintainability

when changeability is?", "most of the characteristics have not been defined at an operational level").

Although the ISO/IEC 9126 quality model has been brought into question by the aforementioned researchers, the principle that within a predefined context, the quality of a software product "can be evaluated by measuring internal attributes (typically static measures of intermediate products), or by measuring external attributes (typically by measuring the behaviour of the code when executed), or by measuring quality in use attributes" [121], is generally accepted and will be followed in this book. However, according to Jørgensen [130], one should avoid considering the measures of the software quality-related attributes (i.e. software quality indicators or factors) as the equivalent of software quality measures.

1.3.3 Software Development Productivity

Munson [190] argues that one of the greatest problems of measurement in SE is the lack of standards for anything we wish to measure e.g. there are no standards for measuring the productivity of programmers. Productivity is usually defined as the output divided by the effort (e.g. measured in hours) required to produce that output [176, 177]. However, the question is how to translate the output into a reasonable measure. According to Maxwell and Forselius [176], output measurement should be based on a project's size, functionality and quality but, unfortunately, such measurement does not yet exist and lines of code (*LOC*) and function point (*FP*) counts are currently the most widely used output measurements. However, *LOC* per unit of effort tends to value longer rather than efficient or high-quality programs. In Object-Oriented (OO) development, a class or method may stand for the unit of output, thus the number of classes per person-year (*NCPY*) and the number of classes per person-month (*NCPM*) are also used as productivity metrics [135]. The number of acceptance tests passed (NATP) was used as an indicator of external code quality by George and Williams [87, 88], Pančur et al. [201], Madeyski [157], as well as Gupta and Jalote [94]. In contrast to some productivity measures (e.g. *LOC* per unit of effort), NATP per unit of effort (e.g. the number of acceptance tests passed per hour) takes into account the functionality and quality of software development products [157, 168] and thus seems to be an interesting productivity indicator. Poppendieck [205] goes even further and argues that to measure the real productivity of software development, we need to look at how efficiently and effectively we turn ideas into software; she proposes to measure the revenue generated per employee. However, we cannot use that measure in non-commercial, academic projects.

The term "software development speed" is sometimes used to avoid misunderstandings related to different meanings of productivity. The research question related to software development speed is posed in Sect. 1.4, while the related research goal is formulated in Sect. 3.1. Software development speed indicators are described in Sect. 3.3.2.3, while the impact of TF on development speed indicator is analysed in Chap. 6.

1.4 Research Questions

The aim of this book is to present the results of an empirical evaluation of the effects of the TF practice on different indicators of software quality, as well as software development speed based on a series of closely related empirical studies. The aim is to shed light on the effects of the TF practice, based on different views, in order to embrace a wider perspective of the possible effects of the TF practice.

Because the readers may benefit from the understanding, at some level, of what is being investigated before they can read how it relates to the already existing work presented in Chap. 2, the research questions are introduced here:

RQ1 What is the impact of TF on external code quality indicator(s)?
RQ2 What is the impact of TF on development speed indicator(s)?
RQ3 What is the impact of TF on internal quality indicator(s)?
RQ4 What is the impact of TF on indicators of the fault detection effectiveness and the thoroughness of unit tests?

As mentioned before, a possible interaction between the TF and the PP practice is also investigated in the first experiment. Hence, the following development techniques are considered: Test-Last Solo Programming (TLSP or TL for short), Test-Last Pair Programming (TLPP), Test-First Solo Programming (TFSP or TF for short) and Test-First Pair Programming (TFPP). The differences between the TF and the TL practices are described in depth in Sect. 3.3.1. Research questions RQ1–RQ4 correspond to Chaps. 5, 6, 7, and 8, respectively.

1.5 Book Organization

This book is divided into ten chapters. The book also contains references to the cited works, glossary, lists of figures, tables, and acronyms, as well as the index of terms.

Chapter 1 starts with an introduction to the investigated TF programming practice and presents the mechanisms behind that practice that motivate empirical investigation. The first chapter also provides background information on research methodology and software measurement used in empirical software engineering. Furthermore, it presents research questions addressed in this book, and gives an overview of the book and its main contributions.

Chapter 2 describes the most important findings from empirical studies concerning the TF and PP software development practices, since both are sometimes used simultaneously.

Chapter 3 presents research goals refined from research questions with the help of a goal definition template, the high level conceptual model that guides the research, the independent, dependent and possible confounding variables.

Chapter 4 focuses on the planning and execution of the three experiments that have been carried out at Wroclaw University of Technology (WUT). The chapter

gives details about the context of the experiments, research hypotheses, experimental materials, tasks and designs. Moreover, there is also a short overview of the new measurement tools (Aopmetrics, Judy, ActivitySensor and SmartSensor plugins) developed for the sake of the experiments carried out.

Chapters 5, 6, 7 and 8 describe the results of the statistical analysis of the impact of the TF practice on the indicators of external code quality, development speed, design complexity, the fault detection effectiveness and the thoroughness of unit tests, respectively.

In order to get more reliable and unbiased conclusions from the merged results of the three performed experiments that address the same research questions, a meta-analysis of the experimental results is presented in Chapter 9. The meta-analysis is based not only on combining p-values, but also on combining effect sizes using fixed, as well as random, effects model.

Chapter 10 provides a summary of the results, presents plausible mechanisms behind the results, main contributions of the book, possible threats to the validity of the results, conclusions and future work.

1.6 Claimed Contributions

The main contribution of this book to the body of knowledge in software engineering consists in the increased understanding of the impact of TF on the percentage of acceptance tests passed PATP (an external code quality indicator), a number of acceptance tests passed per development hour NATPPH (the development speed indicator), internal code quality indicators (i.e. the mean value of: coupling between object classes CBO_{Mean}, weighted methods per class WMC_{Mean} and response for a class RFC_{Mean}), as well as indicators of the fault detection effectiveness (i.e. mutation score indicator MSI) and the thoroughness of unit tests (i.e. branch coverage BC).

Furthermore, practical contributions are related to new measurement tools (ActivitySensor and SmartSensor Eclipse plugins, Judy and Aopmetrics) which have been developed to support the experiments. ActivitySensor and SmartSensor Eclipse plugins contributed to reducing and monitoring several threats to the validity of the results. Judy mutation testing tool contributed greatly to the obtaining of the unique results concerning the impact of the TF practice on mutation score indicator (an indicator of the fault detection effectiveness of unit tests). Aopmetrics helped to collect internal metrics (sometimes labelled as design complexity metrics [238]) from a large number of software projects.

Additional contribution consists in the overview of the state-of-the-art of the empirical studies concerning the effects of the TF and PP practices.