A Software Process Simulation Model of Extreme Programming

Marco Melis

February 7, 2006

Contents

Introduction **vi**

1 XP Overview **1**

2 Software Process Simulation Modeling **6**

2.1 Common uses of simulation modelling 8

2.2 Simulation techniques and approaches 13

2.2.1 Continuous Simulation 13

2.2.2 Discrete Event Simulation 15

2.2.3 Deterministic, Stochastic and Mixed Simulation 20

2.2.4 Sensitivity Analysis 21

3 Studies on XP **22**

3.1 XP Case Studies 22

3.2 Pair Programming 26

3.3 Test First Programming 30

4 Related Works **36**

5 Model Description **38**

5.1 The Simulation Modeling Approach 38

5.2 Model Description 40
5.2.1 Model Entities 41
5.2.2 Model Actors 45
5.2.3 Model Activities 50
5.3 The Simulator Engine 61
5.4 A Quick View to the Simulation Flow 63
5.5 A Better Explanation of the Practices' Usage Level 64
5.6 Calibration and Validation 67

6 Experimental Results **72**
6.1 Research Hypotheses 72
6.2 Simulation Results 73
6.2.1 Further Analysis on the Simulation Model 77
6.2.2 Statistical Analysis of the Results 82

7 Conclusions and Future Work **87**

A Parameters **99**

B Detailed Results **103**

List of Figures

2.1 *Structure of a discrete event simulation system.* 17

2.2 *Details of the event approach structure.* 18

5.1 *Class diagram of the model entities.* 42

5.2 *Class diagram of the model actors.* 46

5.3 *Class diagram of the model activities.* 51

5.4 *Class diagram of the model events.* 52

5.5 *Refactoring probability.* 59

6.1 *Simulation results varying TDD.* 78

6.2 *Simulation results varying PP.* 79

6.3 *Actual PP usage percentage varying PP.* 80

6.4 *Simulation results varying actual PP.* 81

6.5 *Fitting USs distribution.* 82

6.6 *Fitting Days distribution.* 84

6.7 *Fitting Defects/KLOC distribution.* 84

6.8 *Fitting KLOCs distribution.* 85

6.9 *Box plots of the output variables.* 86

List of Tables

3.1 *Measurement of implementation quality.* 24

3.2 *Pair Programming experiments and case studies.* 34

3.3 *TDD experiments and case studies.* 35

5.1 *Defect injection rate multipliers.* 60

5.2 *RepoMargining input parameters.* 68

5.3 *RepoMargining simulation results.* 69

5.4 *M@Info input parameters.* 70

5.5 *M@Info simulation results.* 71

6.1 *The four project research conditions.* 73

6.2 *Simulation results.* 74

6.3 *Hypotheses tests.* 76

6.4 *Simulation results varying problem reports.* 77

6.5 *Test fit for the output variables distribution.* 83

A.1 *Input and output variables.* 99

A.2 *Project specific parameters.* 100

A.3 *Inner model parameters.* 101

A.4 *Practice influence weights.* 102

B.1 *Descriptive Statistics: case 1.* 103
B.2 *Descriptive Statistics: case 2.* 103
B.3 *Descriptive Statistics: case 3.* 104
B.4 *Descriptive Statistics: case 4.* 104
B.5 *Descriptive Statistics: case 1 with PR.* 104
B.6 *Descriptive Statistics: case 2 with PR.* 104
B.7 *Descriptive Statistics: case 3 with PR.* 105

Introduction

Extreme Programming (XP) [4] is a new lightweight software development methodology which has become very popular in recent years. However, the effectiveness of XP practices has still to be assessed. Few studies have been published that provide quantitative results and most of them comes from empirical experiments which are extremely costly to perform and for which it is virtually impossible to achieve a high degree of completeness. The use of simulation to estimate and qualitatively verify the effectiveness of a particular development process often represents a valid alternative solution.

The main goal of this research is to better understand the XP process and to evaluate its effectiveness. In particular it is aimed to investigate how its key practices influence the evolution of a certain project. To achieve this, software process simulation has been chosen. A process model has been developed and a simulation executive has been implemented to enable simulation of XP software development activities. The model follows an object-oriented approach and was implemented in Smalltalk language, following the XP process itself. It is able to vary the usage level of some fundamental XP practices (*Pair Programming*, *Test-First Programming*), and to simulate how the modelled project entities evolve as a result.

Thesis Organization

The thesis is organized as follows:

Chapter 1: this chapter gives a background on the Extreme Programming methodology and of the values and practices which characterize it;

Chapter 2: in this chapter, a description of Software Process Simulation and of the more commons modelling techniques is reported;

Chapter 3: the existing empirical studies on Extreme Programming, with particular attention to those related to Pair Programming and Test-First Programming, are reported here;

Chapter 4: this chapter presents some important related works on simulation modeling of Extreme Programming aimed to study the effectiveness of its practices;

Chapter 6: this chapter shows the experimental results obtained simulating an XP project.

Chapter 7: a summary of the findings of the thesis and further research works are given in this chapter.

Chapter 1

Extreme Programming Overview

In recent years Extreme Programming (XP) [2], [4] is the agile methodology that has received most attention. In XP there is a strong emphasis on informal and immediate communication, automated tests and pair programming, which monitor the progress of the software development, allowing a continuous feedback at different time scales.

The roots of XP lie in the Smalltalk community and, in particular, in the close collaboration of Kent Beck and Ward Cunningham beginning in the late 1980's. Both of them refined these practices on numerous projects during the early 1990's, extending their ideas of a software development approach that was both adaptive and people-oriented [23], [61]. The crucial step from informal practice to a methodology occurred in 1997 when Kent Beck successfully used the full set of XP practices (which will be described later in this section) to implement a payroll project for Daimler Chrysler [23]. Since then, XP has been successfully used at many companies, such as

Bayerische Landesbank, Credit Swiss Life, First Union National Bank, Ford Motor Company and UBS [61].

XP is based on Beck and Cunninghams observations of what made program development faster and what made it slower. XP is an important new methodology because it is one of the several new agile software methodologies created to produce high quality software and reduce the cost of software [23].

These experiences were then formalized and published in 1999 by Kent Beck in the first edition of "*Extreme Programming Explained – Embrace Change*" [2]. He defined a set of 12 practices that embody four fundamental values: communication, feedback, courage, simplicity. Five years later a new edition of the same book appeared, with a renewed vision of the same methodology [4]. This new version is based on the lessons learned by the XP community since the publication of the first book.
The new XP includes five values, fourteen principles, thirteen primary practices and eleven corollary practices. Here I give you a little description of the five values:

Communication This XP value focuses on building a person-to-person, mutual understanding of the problem environment through minimal formal documentation and through maximum face-to-face interaction. XPs practices are designed to encourage interaction, *developer-to-developer* and *developer-to-customer* [12].

Simplicity. This XP value challenges each team member to continuously ask, What is the simplest thing that could possibly work? [12]. The XP originators contend that it is better to do a simple thing today and

pay a little more tomorrow for change than to do a more complicated thing today that may never be used [12].

Feedback. XP teams obtain feedback by testing their software early and often [19]. They deliver the system to the customers as early as possible and implement changes and re-prioritization as suggested by feedback from the customer.

Courage. Developers often cite the pressure to ship a buggy product. It takes courage to resist this pressure [12]. Beck also states that XP teams must be courageous and willing to make changes late in the project or even discard the code and start all over again [12].

Respect. The previous four values imply a fifth: respect. If members of a team dont care about each other and their work, no methodology can work. You must be respectful to your colleagues and their contributions, to your organization, to persons whose life is touched by the system you are writing [40].

As Beck says, "*Values bring purpose to practices.*" and "*Practices are evidence of values*". A brief description list of the primary practices is reported below, as excerpted from "*The New XP*" written by Michele Marchesi [40]:

Stories: the functionalities of the system are described using stories, short descriptions of customer-visible functionalities. Stories also drive system development.

Weekly Cycle: software development is performed a week at a time. At the beginning of every week there is a meeting where the stories to develop in the week are chosen by the customer.

Quarterly Cycle: on a lager time scale, development is planned a quarter at a time. This is made up of reflections on the team, the project and the progress.

Slack: avoid to make promises you cannot fulfill. In any plan, include some tasks that can be dropped if you get behind. In this way, you will keep a security margin, to be used in the case of un-forecasted problems.

Sit Together: development teams should work in an open space, able to host the whole team, to maximize communication.

Whole Team: the team should be composed of members with all the skills and the perspectives needed for the project to succeed. They must have a strong sense of belonging, and must help each others.

Informative Workspace: the workspace should be provided with informative posters and other stuff, giving information on the project status and on the tasks to be performed.

Energized Work: developers must be refreshed, so that they can focus on their job and be productive. Consequently, limit overtime working so everyone can spend time for his or her own private life. This practice in the old version of XP was called "sustainable pace".

Pair Programming: the code is always written by two programmers at one machine. This practice exits already in the original XP.

Incremental Design: XP opposes producing a complete design up front. The development team produces the code as soon as possible in order to obtain feedback and improve the system continuously. Design is

indispensable to obtain good code. The question is when to design. XP suggests to do it incrementally during coding. The way helpful to obtain this is to eliminate duplications in the code.

Test-First Programming: before updating and adding code, it is necessary to write tests in order to verify the code. This solves four problems:

- Cowboy coding: It is easy to get carried away to program quickly and put everything in mind in the code. If we write tests and you have to run them, the tests help us focus on the problem at hand, and can prove that our design is correct.
- Coupling and cohesion: if it isn't easy to write a test, this means that you have a problem of design, not of testing or coding. If your code is loosely coupled and highly cohesive, you can test it easily.
- Trust: if you write code that works and you document it with automated tests, your teammates will trust you.
- Rhythm: it is easy to get lost and wander for hours when you are coding. If you accustom yourself to the rhythm: test, code, refactor, test, code, refactor, it will not happen.

Ten-Minute Build: system should be built and all of the tests should be finished in ten minutes, in order to execute it often and obtain feedback .

Continuous Integration: Developers should be integrating changes every two hours in order to ease integration headaches.

Chapter 2

Software Process Simulation Modeling

Software Process Simulation (SPS) is becoming increasingly popular in the software engineering community, both among academics and practitioners [32]. In fact, new and innovative software engineering techniques are constantly being developed, so a better understanding of these is useful for assessing their effectiveness and predicting possible problems. Simulation can provide information about these issues avoiding real world experimentation, which is both time and cost-intensive.

This area of SPS has attracted growing interest over the last twenty years, but only recently is it beginning to be used to address several issues concerning the strategic management of software development and process improvement support. It can also help project managers and process engineers to plan changes in the development process. The development of a simulation model is a relatively inexpensive way – compared to experimenting with actual software projects – of gathering information when costs, risks and

complexity of the real system are very high.

In order to relate the real world results to simulation results, it is usual to combine empirical findings and knowledge from real processes. In general, empirical data are used to calibrate the model, and the results of the simulation process are used for planning, design and analyzing real experiments [54].

Some key concepts have to be explained in order to have a better understanding of this area, in particular what "model" and "simulation model" means. A **model** is an abstraction (i.e., a simplified representation) of a real or conceptual complex system. A model is designed to display significant features and characteristics of the system, which one wishes to study, predict, modify, or control. Thus a model includes some, but not all, aspects of the system being modelled.

A **simulation model** is a computerized model that possesses the characteristics described above and that represents some dynamic system or phenomenon. One of the main motivations for developing a simulation model or using any other modeling method is that it is an inexpensive way to gain important insights when the costs, risks, or logistics of manipulating the real system of interest are prohibitive. Simulations are generally employed when the complexity of the system being modelled is beyond what static models or other techniques can usefully represent [32].

2.1 Common uses of simulation modelling

Common [1] purposes of simulation models are to provide a basis for experimentation, predict behavior, answer "what if" questions, teach about the system being modelled, etc. A software process simulation model focuses on some particular software development / maintenance / evolution process. It can represent such a process as currently implemented (as-is), or as planned for future implementation (to-be). Since all models are abstractions, a model represents only some of the many aspects of a software process that potentially could be modelled namely the ones believed by the model developer to be especially relevant to the issues and questions the model is used to address.

There is a wide variety of reasons for undertaking simulations of software process models. In many cases, simulation is an aid to decision making. It also helps in risk reduction, and helps management at the strategic, tactical, and operational levels. Here we have a list of the main reasons for using simulations of software processes.

Strategic management. Simulation can help address a broad range of strategic management questions, such as the following. Should work be distributed across sites or should it be centralized at one location? Would it be better to perform work in-house or to out-source (subcontract) it? Would it be more beneficial to employ a product-line approach to developing similar systems, or would the more traditional, individual product development approach be better suited to software

[1] This section has been excerpted from the paper "Software Process Simulation Modeling: Why? What? How?" [32].

process improvement initiatives?

In each of these cases, simulation models would contain local organizational parameters and be developed to investigate specific questions. Managers would compare the results from simulation models of the alternative scenarios to assist in their decision making.

Planning. Simulation can support management planning in a number of straightforward ways, including

- forecast effort / cost, schedule, and product quality;
- forecast staffing levels needed across time;
- cope with resource constraints and resource allocation;
- forecast service-level provided (e.g., for product support);
- analyze risks.

All of these can be applied to both initial planning and subsequent re-planning. Simulation can also be used to help select, customize, and tailor the best process for a specific project context. These are process planning issues.

Control and operational management. Simulation can also provide effective support for managerial control and operational management. Simulation can facilitate project tracking and oversight because key project parameters (e.g., actual status and progress on the work products, resource consumption to-date, and so forth) can be monitored and compared against planned values computed by the simulation. This

helps project managers determine when possible corrective action may be needed.

Project managers can also use simulation to support operational decisions, such as whether to commence major activities (e.g., coding, integration testing). To do this, managers would evaluate current project status using current actual project data and employ simulation to predict the possible outcome if a proposed action (e.g., commence integration testing) was taken then or delayed.

Process improvement and technology adoption. Simulation can support process improvement and technology adoption in a variety of ways. In process improvement settings, organizations are often faced with many suggested improvements. Simulation can aid specific process improvement decisions (such as go / no-go on any specific proposal, or prioritization of multiple proposals) by forecasting the impact of a potential process change before putting it into actual practice in the organization. These applications use simulation *a priori* to compare process alternatives, by comparing the projected outcomes of importance to decision makers (often cost, cycle-time, and quality) resulting from simulation models of alternative processes.

Simulation can also be used ex post to evaluate the results of process changes or selections already implemented. Here, the actual results observed would be compared against simulations of the processes not selected, after updating those simulations to reflect actual project characteristics seen (e.g., size, resource constraints). The actual results would also be used to calibrate the model of the process that was used,

in order to improve future applications of that model.

Just as organizations face many process improvement questions and decisions, the same is true for technology adoption. The analysis of inserting new technologies into a software development process (or business process) would follow the same approach as for process change and employ the same basic model. This is largely because adoption of a new technology is generally expected to affect things that are usually reflected as input parameters to a simulation (e.g., defect injection rate, coding productivity rate) and / or to change the associated process in other more fundamental ways. about software processes in several ways.

Understanding. Simulation can promote enhanced understanding of many process issues. For example, simulation models can help people such as project managers, software developers, and quality assurance personnel better understand process flow, i.e., sequencing, parallelism, flows of work products, etc. Animated simulations, Gantt charts, and the like are useful in presenting simulation results to help people visualize these process flow issues. Simulations can also help people to understand the effects of the complex feedback loops and delays inherent in software processes; even experienced software professionals have difficulty projecting these effects by themselves due to the complicated interactions over time.

In addition, simulation models can help researchers to identify and understand consistent, pervasive properties of software development and maintenance processes (a.k.a. software laws). Moreover, simulations

(especially Monte Carlo techniques) can help people understand the inherent uncertainty in forecasting software process outcomes, and the likely variability in actual results seen. Finally, simulations help facilitate communication, common understanding, and consensus building within a team or larger organization. All simulation models help with process or organizational understanding to some degree.

Training and learning . Simulation can help with training and learning about software processes in several ways. Although this purpose cluster is closely related to that of understanding, the particular setting envisioned here is an explicitly instructional one. Simulations provide a way for personnel to practice / learn project management; this is analogous to pilots practicing on flight simulators. A simulated environment can help management trainees learn the likely impacts of common decisions (often mistakes), e.g., rushing into coding, skipping inspections, or reducing testing time. Finally, training through participation in simulations can help people to accept the unreliability of their initial expectations about the results of given actions; most people do not possess good skills or inherent abilities to predict the behavior of systems with complex feedback loops and / or uncertainties (as are present in software processes). Overall, active participation in a good mix of simulations can provide learning opportunities that could otherwise only be gained through years of real-world experience [32].

2.2 Simulation techniques and approaches

During the design and implementation of a simulator, various techniques and strategies may be adopted to model the behaviour of a given system. Depending upon the system to be simulated, some techniques may be more favourable than others. Factors including the level of abstraction and the desired accuracy and speed of the simulation should be taken into consideration when designing the simulator engine. Traditionally, simulators are designed using either **continuous** or **discrete-event** techniques to simulate a given system [14]. The following sections 2.2.1 and 2.2.2 analyze these different approaches.

Also, it is useful to classify the system being simulated into two separate categories depending upon the degree of randomness associated with the behaviour of the system in its simulated environment. A system that relies heavily upon random behaviour is referred to as a **stochastic** system. Conversely, a **deterministic** simulation system incorporates absolutely no random behaviour whatsoever. As such, the simulation results for a given set of inputs will always be identical [14]. These issues are described in section 2.2.3.

2.2.1 Continuous Simulation

With continuous simulation time is controlled by continuous variables expressed as differential equations. During the simulation the software will integrate the equations.

The more popular approach for simulating in a time continuous way is the System Dynamics modeling. This field was introduced by J. W. Forrester

to apply the engineering principles of feedback and control to social systems [21]. Abdel-Hamid was the first person to use system dynamics for modeling software project management process [1].

In system dynamics a system is defined as a collection of elements that continually interact with each other and outside elements over time, to form a unified whole [28]. The two important elements of the system are structure and behavior. The structure is defined as the collection of components of a system, and their relationships. The structure of the system also includes the variables that are important in influencing the system. The behavior is defined as the way in which the elements or variables composing a system vary over time [28].

System dynamics models describe the system in terms of "flows" that accumulate in various "levels". The flows can be dynamic functions or can be the consequence of other "auxiliary" variables. As the simulation advances time in small evenly spaced increments, it computes the changes in levels and flow rates . For example, the error generation rate may be treated as a "flow" and the current number of errors could be treated as a "level". This allows the model to capture the stability or instability of feedback loops. A system dynamics model can be valuable in finding the levels where a model can become unstable, or in predicting the unanticipated side effects of a change in a system variable [41].

While the system dynamics model is an excellent way to describe the behavior of project variables, it is a more difficult way to describe process steps. While it is possible to represent discrete activities in a system dynamics model, the nature of the tool implies that all levels change at every time

interval. If the process contains sequential activities, some mechanism must be added to prevent all activities from executing at once. For example, if we modelled the software process as define, design, code and test activities, as soon as some code was defined, design would start. If we wanted to model a process that completed all design work before coding started, we would have to create an explicit mechanism to control the sequencing [41].

Because system dynamics models deal with flows and levels, there are no individual entities and thus no entity attributes. For a software process model, this means that all modules and all developers are equal. If we wanted to model the effect of a few error prone code units on the development process, we would not be able to specify which units were error-prone [41].

Finally, a system dynamics model does not easily allow to model the uncertainty inherent in our estimates of model parameters. If we estimate that coding each module will take from 3 to 6 weeks, we would have to translate that estimate into equivalent coding rates. A discrete model could sample from a distribution using a different time for each module. The system dynamics model either must sample the rate at each time step or must use the same rate for each model run [41].

2.2.2 Discrete Event Simulation

Discrete event simulation [20] involves modelling a system as it progresses through time and is particularly useful for analysing queuing systems. Such systems are common in the manufacturing environment and are obvious as work in progress, buffer stocks, and warehouse parts.

A major strength of discrete event simulation is its ability to model ran-

dom events and to predict the effects of the complex interactions between these events. Experimentation is normally carried out using the software model to answer 'what-if?' questions. This is achieved by changing inputs to the model and then comparing the outcomes. This type of simulation is primarily a decision support tool.

Inside the software or model will be a number of important concepts, namely entities and logic statements. Entities are the tangible elements found in the real world, e.g. for manufacturing these could be machines or trucks. The entities may be either temporary (e.g. parts that pass through the model) or permanent (e.g. machines that remain in the model). The concepts of temporary and permanent are useful aids to understanding the overall objectives of using simulation, usually to observe the behaviour of the temporary entities passing through the permanent ones.

Logical relationships link the different entities together, e.g. that a machine entity will process a part entity. The logical relationships are the key part of the simulation model; they define the overall behaviour of the model. Each logical statement (e.g. "start machine if parts are waiting") is simple but the quantity and variety and the fact that they are widely dispersed throughout the model give rise to the complexity.

Another key part of any simulation system is the simulation executive. The executive is responsible for controlling the time advance. A central clock is used to keep track of time. The executive will control the logical relationships between the entities and advance the clock to the new time. The process is illustrated in Figure 2.1. The simulation executive is central to providing the dynamic, time based behaviour of the model. Whilst the

clock and executive are key parts of a simulation system they are very easy to implement and are extremely simple in behaviour.

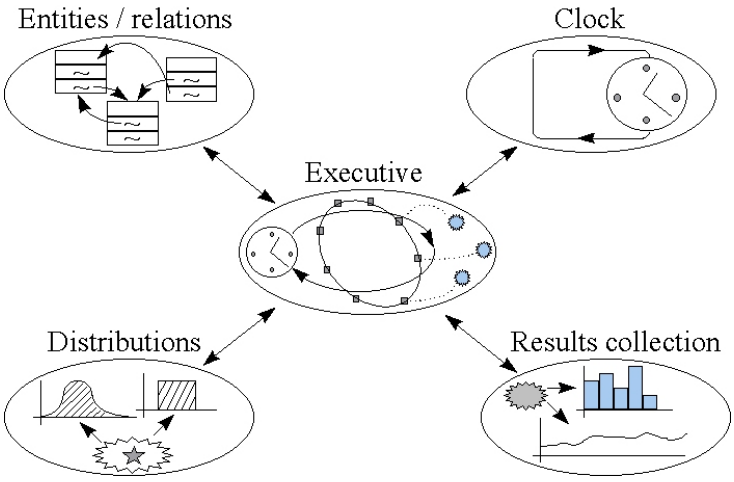


Figure 2.1: *Structure of a discrete event simulation system.*

Two other elements that are vital to any simulation system are the random number generators and the results collation and analysis. The random number generators are used to provide stochastic behaviour typical of the real world.

The model is advanced to the time of the next significant event. Hence if nothing is going to happen for the next 3 minutes the executive will move the model forward 3 minutes in one go. The nature of the jumping between significant points in time means that in most cases the next event mechanism is more efficient and allows models to be evaluated more quickly.

The event approach is described in Figure 2.2. The diagram shows two essential elements: the clock and simulation executive. Here the simulation executive will use an event list (a string of chronologically ordered events). The executive is responsible for ordering the events. The executive removes

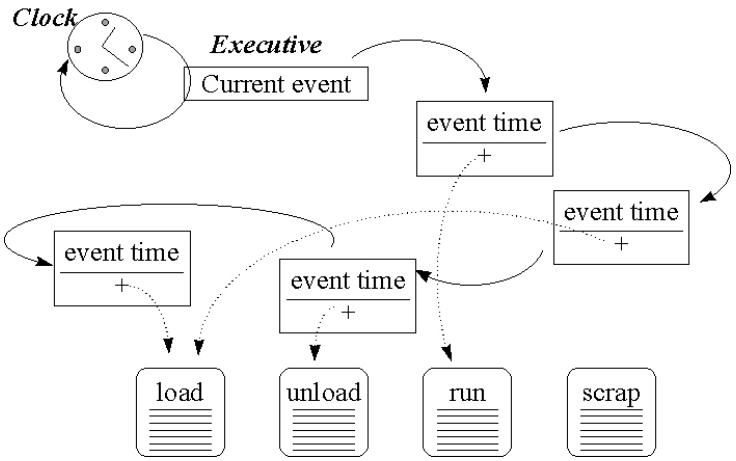


Figure 2.2: *Details of the event approach structure.*

the first event from the list and executes the relevant model logic. Any new events that occur as a result are inserted on the list at the appropriate point (e.g. a machine start load event would generate a machine end load event scheduled for several seconds time). The cycle is then repeated.

Each event on the event list has two key data items. The first item is the time of the event which allows it to be ordered on the event list. The second item is the reference to the model logic that needs to be executed. This allows the executive to execute the correct logic at the correct time. Note that more than one event may reference the same model logic, this means that the same logic is used many times during the life of the simulation run [20], [35], [48], [55].

Discrete event simulation is efficient and particularly appealing when the process is viewed as a sequence of activities, such as in a manufacturing line where items or entities move from station to station and have processing done at each station. Discrete models easily represent queues and can delay processing at an activity if resources are not available. In addition, each

entity may be described by unique "attributes". Changes to the attributes by the activities can provide much of the value of a discrete model.

Because a discrete model allows each entity to contain unique values for attributes, the model can capture the variation in code difficulty and programmer capability. The effects of increased complexity on effort and error rates or the impact of different programmer capabilities on coding duration may be computed from attributes in a discrete model.

Values for attributes may be constant or may be sampled from distributions. The stochastic nature of the variables captures the uncertainty present in measuring the attributes.

Finally, a discrete model allows us to represent the interdependence that occurs between activities in a project. Activities in a development process may be delayed when a programmer is diverted to another task. Testing may be delayed until a test bed is released. If a model can capture these dependencies at a sufficiently detailed level, it may show ways to alter the process to reduce risk or increase efficiency [41].

As mentioned above, a discrete model advances time only when an event occurs. This means that continuously changing variables are only updated at the event times. While the time between discrete events may be days or weeks in a software project, the model of the continuous variables may require a time step in hours. This difference can cause errors in the integration of the continuous variables or may create instability in the behavior of feedback loops.

In addition, because discrete models are based on the idea of a sequence of activities, it may be awkward to represent simultaneous activities. While

activities can occur in parallel, it is difficult to represent the idea of an entity in two activities simultaneously. Imagine a code module in which some parts are in coding while other parts are in unit test. To capture this in a discrete model, we are forced to model sub-components of the module so that each sub-component can be in only one activity at a time [41].

2.2.3 Deterministic, Stochastic and Mixed Simulation

A simulation can be deterministic, stochastic, or mixed. In the deterministic case, input parameters are specified as single values (e.g., coding for this unit will require 5 work-days of effort, or 4 hours per hundred lines of code; there will be two rework cycles on code and unit test; etc.). Stochastic modeling recognizes the inherent uncertainty in many parameters and relationships. Rather than using (deterministic) point estimates, stochastic variables are random numbers drawn from a specified probability distribution. Mixed modeling employs both deterministic and stochastic parameters.

In a purely deterministic model, only one simulation run is needed for a given set of parameters. However, with stochastic or mixed modeling the result variables differ from one run to another because the random numbers actually drawn differ from run to run. In this case the result variables are best analyzed statistically (e.g., mean, standard deviation, distribution form) across a batch of simulation runs; this is termed **Monte Carlo simulation**. Although many process simulation tools support stochastic models, only some of them conveniently support Monte Carlo simulation by handling batches well.

2.2.4 Sensitivity Analysis

Finally, sensitivity analysis is a very useful technique involving simulation models. Sensitivity analysis explore the effects on the key result variables, of varying selected parameters over a plausible range of values. This allows the modeler to determine the likely range of results due to uncertainties in key parameters. It also allows the modeler to identify which parameters have the most significant effects on results, suggesting that those be measured and / or controlled more carefully. As a simple example, if a 10% increase in parameter A leads to a 30% change in a key result variable, while a 10% increase in parameter B leads to only a 5% change in that result variable, one should be somewhat more careful in specifying or controlling parameter A. Sensitivity analysis is applicable to all types of simulation modeling. However, it is usually performed by varying the parameters manually, since there are few simulation tools that automate sensitivity analysis.

$$S_x^y = \lim_{\Delta x \to 0} \frac{\Delta y / y}{\Delta x / x} = \frac{\partial y}{\partial x} \frac{x}{y} \tag{2.1}$$

Chapter 3

Studies on XP

This chapter aim to present the existing studies on Extreme Programming, with particular attention to those related to Pair Programming and Test-First Programming.

3.1 XP Case Studies

A case study by Poole and Huishman [50] at IOANA Technology reported the productivity improvement when XP was used for maintenance work. The practices of XP were applied to deal with the maintenance of the older version of one of their product. They achieved 67% productivity improvement in terms of bugs fixed per iteration. The productivity is based on a constant work force with no change in overall work habits in terms of hours spent fixing bugs.

Maurer and Martel [42], [43] reported another case study in which they studied a small company with nine full-time programmers developing a Web-based system over 16 months. The first version of the system (first 9 months)

was developed using a document-centric development process. For the second version of the system (last 5 months), the company switched to XP. They calculated four productivity metrics from one release to the next: newly created lines of code ($NLOC$) per unit effort, number of new methods per unit effort, number of new classes per unit effort, number of fixed bugs and new features in a given release per unit effort. A gain in productivity was found for the first three metrics: +63% for $NLOC/effort$, +302% for $\#methods/effort$ and 283% for $\#classes/effort$. On the other hand, the last metric $(\#bugs + \#features)/effort$ showed a slight productivity decrease (-5%) after adopting agile practices.

In addition they inspected three other metrics aimed to understand the average size of the new code added per closed feature or bug fix:

- $NLOC/(\#bugs + \#features)$
- $\#methods/(\#bugs + \#features)$
- $\#classes/(\#bugs + \#features)$

It was found a large increase in these three metrics after adopting XP (+74%, +323% and +303% respectively). Assuming that program size is somehow correlated with problem difficulty, these results show that during the XP timeframe the team simply tackled more complex problems. This interpretation explains the decrease in the last productivity metric.

Another case study was conducted by Hodgetts and Philips [26] at Escrow.com, an internet business-to-business startup. During the same project, the first version of the system was developed using a traditional defined process (V2) for about 20 months. Then the company switched to XP and

a new version of the system (V3) was developed in 12 months. The V2 project employed 21 developers during its existence, with a total cost of 207 developer-months. The V3 team was made up of 4 developers, employing a total cost of 40 developer-months. These results represent an 80% reduction in development-month effort. However, the V3 team was staffed by senior developers, and their expertise likely contributed to the productivity gains.

Switching to XP brought a reduction in the release cycle from two months to 2 weeks, enabling product managers to flexibly and quickly respond to changing business conditions.

Also, XP allowed to reach an increased product quality due to the automatization of the acceptance testing process. They showed a 70% reduction in the number of defects and, in addition, they found a lower defect severity level.

Another advantage of the adoption of XP was in the increased quality of design and implementation. For measure these entities they used some simple indicators: total code size, average size of classes, average size of metrics, and the average cyclomatic complexity of methods. These results are reported in Table 3.1.

Table 3.1: *Measurement of implementation quality.*

Metric	V2 (non-XP)	V3 (XP)	%Change
Total code size	45773	15048	-67%
Average methods per class	6.30	10.95	+73%
Average LOC per method	11.36	5.86	-48%
Average cyclomatic complexity	3.44	1.56	-54%

The reduction in code size is indicative of a simpler implementation, as-

suming delivery of comparable functionality. The presence of a larger number of smaller methods per class, combined with the reduced complexity of methods, suggest an improved division of responsibility and behaviour across methods.

Wood and Kleb [65] conducted a pilot project in order to assess the XP adoption at the NASA Langley Research Center. The project consisted of two release cycles, each subdivided into three two-week iterations, for a total project length of 12 weeks. Their results indicated that the XP approach was approximately twice as productive as similar historical projects they had undertaken in the past. They implemented functionalities at the historical rate, but also supplied a large amounts of supporting tests. Furthermore, the production code was about half as many lines of code as expected and the code readability was much improved, mainly due to continuous refactoring, emergent design and constant code review.

Drobka et al. [17] used XP with four development teams inside their company over an 18-month period. They qualitative and quantitative measured a number of process variable in order to investigate how the XP adoption could change their initial waterfall based process. They conducted a survey by which it was found that XP increased the team's morale and shortened the initial project-learning curve. Measuring the productivity, in terms of assembler-equivalent lines of code per unit effort, they found a high productivity improvement related to previous waterfall projects.

3.2 Pair Programming

Pair Programming practice states that any production code must be created by a pair of developers working together at one computer on the same task. The claimed benefits are diverse:

- it helps to implement the system quickly by speeding up the program development and the bugs removal;
- continuous review is made by the pair developer, so many bugs are removed at the earliest possible point in the software development cycle where they have the least impact on schedule and cost;
- it leads to a better distribution of knowledge among the team members, because everybody works with everybody else during the project, each time on probably different part of the system. It is also a good way to introduce new team members to the project;
- learning is improved, because each member of the developer contributes with her/his personal experience, information and skills and, on the other hand, learns from her/his pair mate experience, information and skills;
- a further benefit is the improved design. While one partner is busy typing or writing down the code, the other partner can think more strategically about the implications of the design and can consider alternative solutions.

However, software developer managers often reject the use of pair programming inside their organization. This is because they assume there will

be a 100 percent programmer-hour increase by putting two developers on a job that only one can do. If pair programming does, indeed, double the time, it certainly would be difficult to justify [62].
Here I report some quantitative studies conducted to assess the validity and efficiency of this practice. A summary of all these findings has been reported in Tab. 3.2.

One of the first experiments conducted to assess the efficiency of collaborative working on software development comes from the Hill Air Force Base in 1975: the so called "Two-Person Team" (2P) [29]. The two-person team approach places two engineers or two programmers in the same location with one workstation and one problem to solve.
The project was a real-time, multitasking system executive of approximately 30,000 Fortran source lines. The team is not allowed to divide the task but produces the design, code, and documentation as if the team was a single individual. Final project results were astounding. Total productivity was 175 lines per person-month (lppm) compared to a documented average individual productivity of only 77 lppm prior to the project (+127%). This result is especially striking when we consider two persons produced each line of source code (144 lppm). The error rate through software-system integration was three orders of magnitude lower than the organizations norm.
Why were the results so impressive? A brief list of observed phenomena includes focused energy, brainstorming, problem solving, continuous design and code walkthroughs, mentoring, and motivation.

An experiment conducted by Nosek at Temple University [47] studied 15 full-time, experienced programmers working for 45 minutes on a challenging

problem. Five of them worked individually, while the others worked collaboratively in five pairs, at the same conditions. The pairs completed the task 40% more quickly and effectively by producing better algorithms and code in less time.
The results found by this experiment have shown statistical significance. Unfortunately little can be generalized because the experiment only involved the completion of a single 45 minute task.

Laurie Williams – University of Utah – conducted a controlled experiment on 28 students working in pair programming and 13 students in individual programming [13], [63] and [62]. She found that paired programmers spent, on average, 15% more time to complete two projects than the solo programmers spent to complete just one, suggesting that pair programming is 40-50% faster than solo programming.
Moreover, the pairs passed, on average, 15% more of the automated post-development test cases and implemented the same functionalities in more than 20% fewer lines of code than their individual counterparts. Implementing functionality in fewer lines of code is commonly viewed as an indication of better design quality and lower projected maintenance costs [5].

Nawrocki and Wojciechowski [46] conducted a similar experiment at the Poznan University of Technology. A group of 21 students was divided into 3 sub-groups: 10 programmers worked in pairs following an XP-like process (XP2); 5 students worked individually using an XP-like process too (XP1), while the last 6 programmers worked individually following the Personal Software Process approach (PSP). They were asked to write four small programming tasks. The results showed that there is no significant difference

in the average development time between pair programming (XP2) and individual programming (XP1). On the other hand the standard deviation of the development times and program sizes is lower for the pair programming group (XP2), suggesting that pair programming is more predictable than individual programming.

Vivekanandan [59] performed a controlled experiment with 214 students: 116 worked in pairs and 98 individually. The experiment consisted in doing a short programming task of about 3 hours and relative debugging to satisfy a defined set of test cases. He found that the PP group employed 19% less time than individuals to complete the same assignment. In other words, pairs were 23% faster than individuals. Also, 8% better design and an improvement of 8% in knowledge and programming skills were achieved by the PP group.

Padberg and Muller [49] used a combination of different metrics to evaluate the cost and benefits of Pair Programming using an economic model for the business value of a development project. They found that pair programming can increase the value of the project when time to market is the decisive factor, and that programmer pairs are faster than single developers.

Vivekanandan [59] performed a controlled experiment with 214 students: 116 worked in pairs and 98 individually. The experiment consisted in doing a short programming task of about 3 hours and relative debugging to satisfy a defined set of test cases. He found that the PP group employed 19% less time than individuals to complete the same assignment. In other words, pairs were 23% faster than individuals. Also, 8% better design and an improvement of 8% in knowledge and programming skills were achieved by the PP group.

Hulkko and Abrahamsson [27] conducted four case studies on the impact of pair programming on software productivity and quality. In contrast to other researchers, they found that Pair Programming does not provide any proven quality benefits and does not result in consistently superior productivity when compared to solo programming.

3.3 Test First Programming

In Test-First Programming practice, also known as Test Driven Development, the developers have to write the test before the code itself. Through a rapid cycle of adding new tests, making them pass, and then refactoring to clean code, the software design evolves through the tests. Also, all tests should be automated [15]. From here on, I will refer to this practice as TDD. Development, the developers have to write tests before the code itself. Through a rapid cycle of adding new tests, making them pass, and then refactoring to clean code, the software design evolves through the tests. Also, all tests should be automated [15]. From here on, I will refer to this practice as TDD.

Many studies reported in the literature have investigated the benefits of TDD. A summary of all of these findings has been reported in Tab. 3.3. benefits of TDD. A summary of all these findings has been reported in Tab. 3.3.

Ynchausti [66] presented a study where unit testing was integrated into the software development process of a five-member programming team using a test-during-coding training module. Individual and pair developer performance was measured before and after the training module was presented. The improvements in quality achieved by the team ranged from 38% to 267%

fewer defects. On the other hand, test-during-coding process took more time to implement the assignment (from 60% to 100%), and led to an increase in size of about 100% more code in the form of unit tests.

George and Williams [24] carried out an experiment with 24 professional pair programmers. The programmers were divided into two groups, one developed code using TDD while the other one used a waterfall-like approach. The experiment showed that the TDD developers took more time (16%) than those not using this practice, but produced higher code quality. In fact this code passed 18% more functional black box test cases than the one developed without applying TDD. However, the variance in the performance of the teams was large and these results are only directional. Additionally, the control group pairs did not generally write any worthwhile automated test cases (though they were instructed to do so), making the comparison uneven. Ynchausti [66] presented a study where unit testing was integrated into the software development process of a five-member programming team using a test-during-coding training module. Individual and pair developer performance was measured before and after the training module was presented. The improvements in quality achieved by the team ranged from 38% to 267% fewer defects. On the other hand, test-during-coding process took more time to implement the assignment (from 60% to 100%), and led to an increase in size of about 100% more code in the form of unit tests.

Muller and Hagner [45] conducted a controlled experiment with 19 computer science graduate students with a median programming experience of 8 years, in order to compare TDD with the traditional development process. Students were divided into two groups and were assigned the same

programming task. They investigated the effectiveness of TDD in terms of development time, reliability and understandability. They found that this XP practice neither leads to quicker development nor does it enhance quality but it would appear to support better program understanding.

George and Williams [24] carried out an experiment with 24 professional pair programmers. The programmers were divided into two groups, one developed code using TDD while the other one used a waterfall-like approach. The experiment showed that the TDD developers took 16% more time than those not using this practice, but produced higher code quality. In fact this code passed 18% more functional black box test cases. However, the variance in the performance of the teams was large and these results are only directional. Additionally, the control group pairs did not generally write any worthwhile automated test cases (though they were instructed to do so), making the comparison uneven.

Williams et al. [64] conducted a case study to investigate the effects of TDD at IBM. They found that the introduction of this practice achieves about a 40% reduction in the defect density of functional verification tests. Also they reported that the team introduced more than 0.5 lines of test code for every line of implementation code. While the developers spend more time writing the test cases, TDD reduces the time they spend in debugging and the overall productivity was roughly the same in both cases.

Erdogmus et al. [18] conducted a controlled experiment with undergraduate students divided into two groups. The experiment group (Test-First) developed using TDD, while the control group (Test-Last) wrote tests after coding. They reported that Test-First programmers wrote more tests

per unit of programming effort than Test-Last group. They observed that writing more tests led to a higher level of productivity and improved the minimum quality achievable.

Table 3.2: *Summarized results for Pair Programming experiments and case studies.*

Authors	Description	Results (*PP vs noPP*)
Nosek, 1998 [47]	Controlled experiment with 15 professionals - 5 pairs *vs* 5 individuals - 45 minute programming task	PP was 40% faster.
Williams et al., 2000 [13, 63, 62]	Controlled experiment with 41 students - 14 pairs *vs* 13 individuals - 4 programming tasks	PP was 40-50% faster, passed 15% more functional tests, wrote more than 20% fewer lines of code.
Nawrocki and Wojciechowski, 2001 [46]	Controlled experiment with 15 students - 5 pairs *vs* 5 individuals - 4 programming tasks	No significant difference in terms of development time.
Lui and Chan, 2003 [38]	Controlled experiment with 15 professionals - 5 pairs *vs* 5 individuals - logical and deduction problems tasks	PP reached 85% of correctness and the individuals only achieved 51%, but PP spent much more time. To reach the same quality, a pair spent 4.2% less time than did individuals on the same tasks.
Vivekanandan, 2004 [59]	Controlled experiment with 214 students, 58 pairs *vs* 98 individuals - 3 hours programming tasks + debugging	PP was 23% faster, gave 8% higher design, gave 8% higher knowledge and programming skills.
Hulkko and Abrahamsson, 2005 [27]	4 case studies with both professionals and students - 8 weeks projects	No significant difference in terms of quality and productivity.

Table 3.3: *Summarized results for TDD experiments and case studies.*

Authors	Description	Results (*TDD vs noTDD*)
Ynchausti, 2001 [66]	Case study with 5 professionals developing 2 small programs, TDD *vs* Test Last.	TDD took more time (60 - 100%), led to 100% more code, gave 38% - 267% fewer defects.
Muller and Hagner, 2002 [45]	Controlled experiment with 19 students, divided into two groups performing the same programming task.	TDD seems to support better program understanding. No significant difference in terms of quality and productivity.
George and Williams, 2003 [25]	Controlled experiment with 24 professionals - 12 using TDD *vs* 12 not using TDD.	TDD took more time (16%), passed 18% more functional black box test cases.
Williams et al., 2003 [64]	Case study in industrial environment.	TDD gave 40% lower defect density. No productivity gain.
Erdogmus et al., 2005 [18]	Controlled experiment with students. TDD *vs* Test Last.	TDD led to a higher level of productivity and improved the quality.

Chapter 4

Related Works

In spite of the widespread use of Extreme Programming (XP) in academic and industrial spheres, only recently have the first attempts been made to simulate XP processes, all using the System Dynamics approach. Some significant contributions are mentioned here.

In [12] Cao proposes a system dynamic simulation model to investigate the applicability and effectiveness of agile methods and to examine the impact of agile practices on project performance in terms of quality, schedule, cost and customer satisfaction. The paper does not provide any quantitative or qualitative results.

Misic et al. [44] investigate the possibility of using system dynamics to model, and subsequently simulate and analyze, pair programming practice and pair switching. The model allows the exploration of some variables affecting pair programming efficiency. The authors find that XP proficiency increases with both psychological compatibility and pair adaptation speed between the members of the pair. Also, the authors observe that the XP process appears to have an advantage over the traditional approach when

pair switches are not too frequent.

Two of the most significant works are those conducted by Kuppuswami et al. [37], [36]. In [37], they propose a system dynamics simulation model of the XP development process to show the constant nature of the cost of change curve that is one of the most important claimed benefits of XP. They also describe the steps to be followed to construct a cost of change curve using the simulation model. In [36], they developed a system dynamics simulation model to analyse the effects of all the XP practices on the software development effort. The developed model was simulated for a typical XP project of 50 User Stories size and the effects of all the individual practices were computed. The results indicated a reduction in software development cost by enhancing the usage levels of individual XP practices.
However, not enough information was provided about the validation of the developed simulation models.

Chapter 5

Model Description

The present chapter gives a detailed description of the model developed in the context of this research thesis.

5.1 The Simulation Modeling Approach

To gain a greater insight into the XP process it was chosen to develop the whole simulation model following the XP methodology ourselves. The system was developed using Smalltalk object oriented language. This choice fell on this language for a number of reasons. First of all, Smalltalk is one of the first object oriented languages, which has been giving support to simulation since its beginning [34], [58], [67]. It is easy to use, powerful and achieves high development productivity levels. In addition, Smalltalk is closely related with Extreme Programming. In fact, Kent Beck, Ward Cunningham, Ron Jeffries [53], Alistair Cockburn [52] and other XP-evangelists, have used Smalltalk as their preferred programming language [2], [10], [9]. Also, many XP support tools were first developed in Smalltalk, such as the xUnit testing

framework [11], [3] and the refactoring browser [51]. Furthermore, object orientation techniques provide a set of values, such as reusability, adaptability and maintainability, which are very desirable for simulation developers [67].

As regards the specific simulation technique, I followed a hybrid approach merging Discrete Event Simulation (DES) and System Dynamics (SD) simulation so as to exploit the advantages of both.
One of the main advantages of the DES approach is that it allows to easily define a stochastic model and perform Monte Carlo simulations, taking into account the intrinsic risk and uncertainty of real projects. For example, the estimated effort required to implement each user story could be better modelled using an appropriate random distribution.

Another reason for choosing DES is that in this way each model entity is well identified and characterised by a number of attributes whose values may change when some specific events are executed. So, we can examine the status of each model entity at each time step of the simulation, gaining a better understanding of the evolution of the process during a simulation run.

On the other hand, with DES it is not easy to model complex feedback loops or aspects of the system that vary continuously. Thus, some aspects of the SD approach were also used, albeit to a lesser extent.

The the two techniques are merged using integration rates (typical of System Dynamics) and updating entity attributes at time steps driven by event execution (Discrete Event). A specific case is the rate at which a developer implements a specific user story: it increases with her/his skill level that varies continuously with her/his cumulated experience on the project. Nevertheless, this model updates developers' experience level at discrete steps,

at the end of each development session.
More details about hybrid simulation techniques can be found in [41], [60].

5.2 Model Description

The model is characterized by several activities: ReleasePlanning, ReleaseImplementationPhase, IterationDevelopment and DevelopmentSession. The inputs to these activities are entities that are modified and created by other instances of activities.

Each activity is executed by one or more actors in the process. Each actor has some attributes, which vary in time, and can perform a number of actions. For example each developer is characterized by experience, skill and velocity.

The model development was based on the following assumptions:

- User Stories are independent from each other. For this reason, the code needed to implement a specific user requirement is associated to one and only one User Story.
- User Stories are not split into engineering tasks, as normally happens in an XP approach. It was not considered necessary for the scope of this model.
- The development team remains the same throughout the project. Turnover was not considered.
- Problems/defects are only found and committed at the end of a release.

- Debugging activities do not add any other code and defects to the system.

The time granularity used in the model is of the duration of a DevelopmentSession, which typically lasts a couple of hours. The equations regulating the model entity variations and the execution of each activity have been taken from existing models, empirical data and, when necessary, from authors assumptions.

5.2.1 Model Entities

As shown in Fig. 5.1, in the model we have 5 concrete type of entities: Project, Release, Iteration, UserStory and ProblemReportStory. Project, Release and Iteration are also kind of ProcessEntity, a common superclass that takes into account of their common features, such as *name*, *startTime* and *stories*, a collection of UserStories. A Project contains a set of Releases, and each Release has a set of associated Iterations. A ProblemReportStory is a specialization of UserStory. A detailed description of such entities is given below.

Entity: User Story

UserStories are the fundamental documents of an XP process. Basically, a UserStory (US) represents a single requirement specification of the system, written by the customer or system user.

As shown in the class diagram in Fig. 5.1, a US is characterized by the following set of attributes:

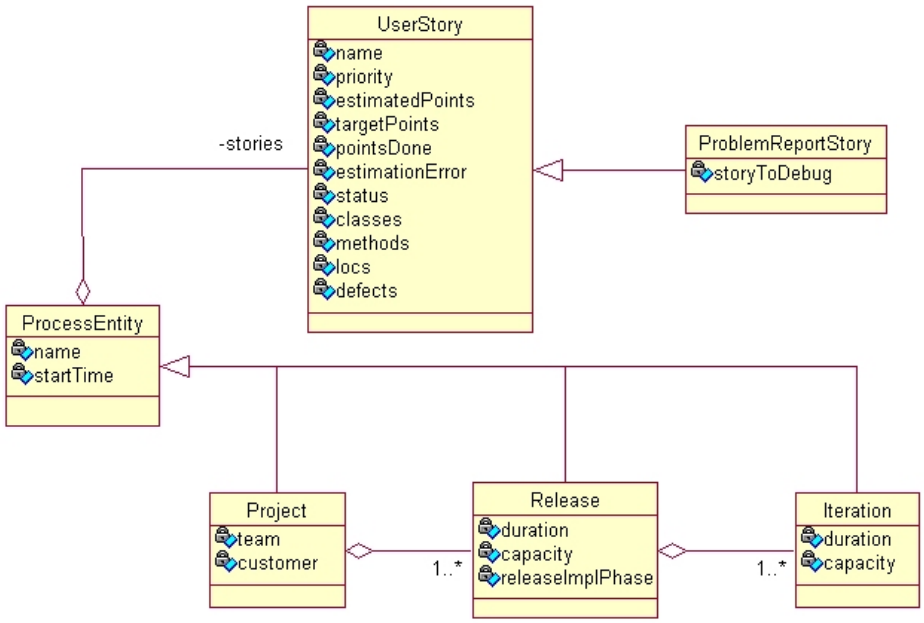


Figure 5.1: *Extract from the class diagram of the identified XP high-level entities.*

- *priority*, which is the business value assigned by the customer using three possible values: *Must*, *Should* and *Could*. One of these three values is assigned to each story with equal probability.

- *estimatedPoints*, whose value represents the estimated effort – in terms of story points – for each UserStory (values taken from a log-normal distribution).

- *estimationError*, indicating the error of each estimation (values taken from a log-normal distribution).

- *targetPoints* (see equation 5.1), whose value represents the actual effort needed to complete a UserStory.

$$TargetPoints = (1 + EstimationError) \cdot EstimatedPoints \quad (5.1)$$

- *pointsDone*, that is the actual effort updated at the end of each DevelopmentSession.
- *status*, which represents the actual working state of the UserStory (four possible values: *ToDo*, *InProgress*, *Done* and *NotCompleted*).
- *classes, methods, locs*, which represent the number of classes, methods and source lines of code developed for implementing the specific UserStory.
- *defects*, the number of defects injected during code implementation.

Entity: ProblemReportStory

This entity is a particular kind of UserStory, which is used to report problems and bugs found in the system by the Team or the Customer. Each ProblemReportStory (PR-US) is scheduled as the other UserStories, but its implementation can be considered as a debugging activity in that aimed to solve problems or fix bugs.

A ProblemReportStory inherits all the attributes and operations of a simple UserStory. In addition, each PR-US has an associated UserStory (*storyToDebug*) since it has been made the assumption that each problem/bug is associated to only one specific implemented UserStory.

Another assumption is that when a PR-US is implemented, no further bugs are injected into the system and no code are added or removed. The only effect of a PR-US implementation is to reduce the number of defects associated to the *storyToDebug*.

The mechanism through which the defects are reduced depends on a

stochastic rule. In particular, each defect of the *storyToDebug* is fixed with a probability P_{fix} given as an input model parameter.

When TDD is used, the debugging time to fix a single defect decreases. This assumption is based on the fact that the use of TDD lead to an higher test coverage for the system then, quite likely, it decreases the time needed to fix defects. Therefore, it is plausible that the debugging activities are more efficient. This is in agreement with a case study conducted by Lui et al. [39] in which they found that non-TDD teams were able to fix only 73% of their defects against 97% fixed at the same time by TDD-Teams.

In accordance to these results, when TDD is not used, P_{fix} has been set to $P_{fix}|^{min} = 0.73$ and linearly increase with the adoption level of this practice ($A_{\%}^{Tdd}$), with a maximum of $P_{fix}|^{max} = 0.97$.

Entity: Project

This entity represents a software project. It is characterized by a *name*, a *startTime*, a *team* and a *customer*. It also have an attribute called *stories* which represents the set of UserStories still to implement. The development of a Project goes through a number of subsequent Releases of the system (attribute *releases*).

Entity: Release

An XP Release represents a predefined period of time in which a set of User Stories, chosen by the Customer, have to be implemented to deploy a new version of the system under development.

Each Release has its own identifier (*name*), a *startTime*, a *duration*, a

capacity, a number of ITERATIONS (*iterations*) and an associated activity: *ReleaseImplementationPhase*. Each RELEASE contains a set of USERSTORIES (*stories*).

Entity: Iteration

XP follows an iterative approach to development, using time-boxed cycles. Each Release of the system is implemented through a predefined number of time boxed Iterations. An ITERATION has its own identifier (*name*), a *startTime*, a *duration*, a *capacity* and, again, contains a set of USERSTORIES (*stories*).

5.2.2 Model Actors

Each activity is executed by one or more actors in the process. The identified actors are the CUSTOMER and the DEVELOPERS making up the TEAM (see Fig. 5.2).

Actor: Team

The TEAM can be considered as a system actor, although it is made up of other actors (DEVELOPERS). In fact, the TEAM can perform actions, activities and modify entities' attributes.

As said before, the TEAM is composed by a number of DEVELOPERS or, in other words, it has an attribute called *developers* which is a collection of DEVELOPERS. It is responsible for the creation of these actors when a new project simulation starts. The number of developers is given as an input of the simulator and chosen by the simulator user, such as the other input

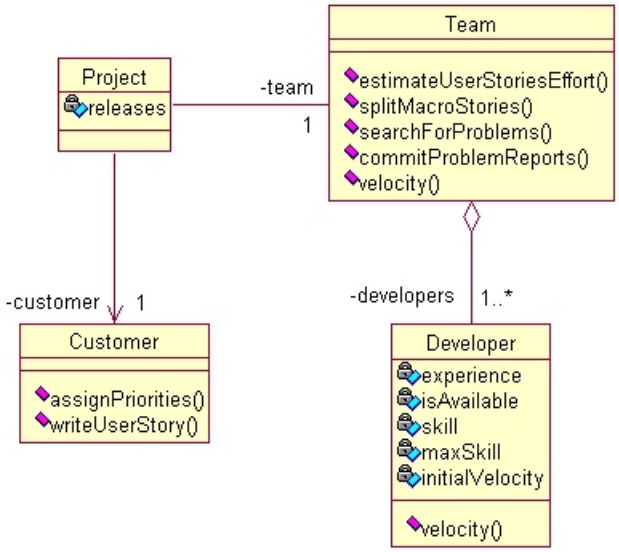


Figure 5.2: *Extract from the class diagram of the identified XP actors.*

parameters. One of these is the *initialTeamVelocity* that is a number specifying the typical speed at which a specific TEAM can implement a software project, in terms of Story Points[1] per day. The created DEVELOPERS have their own *velocity* which vary during a project. So, the TEAM *velocity* (V_{team}) vary during a project simulation, and is given by the sum of the individual DEVELOPERS' *velocities*, as shown in the Equation 5.2.

$$V_{team}(t_i) = \sum_j V_{D_j}(t_i) \tag{5.2}$$

The TEAM is also responsible for:

- estimating the effort needed to implement each USERSTORY;
- splitting the USERSTORIES that are too big (Macro USERSTORIES)

[1] An XP Story Point is a relative unit used by XPers to estimate and measure efforts. Frequently, a Story Point is intended as an ideal working day of development, but each team can use its own definition.

into smaller ones;

- searching problems in the implemented system and tracking them (ProblemReportStories).

The process of estimating the UserStories is performed during the ReleasePlanning activity. For each UserStory inside the Project's *stories* container, the estimated effort (*estimatedPoints*) is drawn from a log-normal distribution with given descriptive parameters (mean and standard deviation). These distribution parameters are given as inputs to the simulator and are taken from historical data of other company's projects or from the same project under simulation whether it is already started.

I suppose that a Team can implement at least two ($K_{macro} = 2$) UserStories for each week. For this reason, the UserStories whose estimation is greater than half the week's capacity[2] are classified as "Macro UserStories", then split by the team into smaller ones. The limit identifying a Macro UserStory (K_{macro}) can be changed in accordance to the specific company's needs (see Tab. A.3).

Upon completion of each Release the Team reports any problems encountered in the system developed so far. The number of problems is proportional to the system defect density in terms of residual defects per US. For each problem found, a ProblemReportStory is created and planned in the subsequent releases as the other UserStories.

[2]The week's capacity is given by the daily team velocity (V_{team}) multiplied by the number of working days on a week.

Actor: Developer

Developers are the main actors of an XP software development process. They are responsible for develop the whole system starting from the USs defined by the Customer. In the model, each DEVELOPER is characterized by a set of attributes: *experience*, *skill*, *maxSkill*, *initialVelocity* and *isAvailable*.

Experience is the cumulated working days by each DEVELOPER on the project and is updated at the end of each DEVELOPMENTSESSION, as formally expressed by the following equation:

$$E(t_i) = E(t_{i-1}) + \Delta t_{sess}(t_i) \tag{5.3}$$

where $E(t_{i-1})$ is the *experience* cumulated by the DEVELOPER at the time t_{i-1}, just before starting the i^{th} DEVELOPMENTSESSION; $E(t_i)$ is the DEVELOPER's *experience* after having completed the i^{th} DEVELOPMENTSESSION, while $\Delta t_{sess}(t_i)$ is the duration of the i^{th} DEVELOPMENTSESSION.

The *skill* attribute represents the DEVELOPER's ability to develop software and is judged on a scale from $S_{min} = 1$ to $S_{max} = 10$. Each DEVELOPER instance is created with an initial *skill* value which is randomly chosen from 1 to 3. Then, the individual *skill* linearly increases with the *experience* gained on each DEVELOPMENTSESSION with a coefficient K_{learn} (learning coefficient).

$$skill(t_i) = skill(t_{i-1}) + \Delta E(t_i) \cdot K_{learn} \tag{5.4}$$

Also, each DEVELOPER can attain a maximum skill level (*maxSkill*), which differs from one to another, randomly chosen from 7 to 10. This takes into account the fact that a DEVELOPER's capability is also affected

by her/his individual characteristics. In fact, a DEVELOPER with a lower initial *skill* rating can equal and surpass her/his team mate's *skill* level.

A value of K_{learn} was chosen such that a DEVELOPER with the minimum initial *skill* level ($S_{min} = 1$) can attain the maximum *skill* level ($S_{max} = 10$) after about 5 years[3].

$$S_{max} = S_{min} + 1000 \cdot K_{learn} \quad \Longrightarrow \quad K_{learn} = \frac{S_{max} - S_{min}}{1000} = 0.009$$

Based on empirical findings (see Section 3.2) a different value for the learning coefficient K_{learn} was used depending on whether the session was to be performed in pair programming or not. In particular, the skill increment is 8% higher for a pair programming session [59]. For this reason we have two different values for the learning coefficient:

$$K_{learn} = \begin{cases} 0.009 & \text{solo session,} \\ 0.00972 & \text{pair programming session.} \end{cases}$$

A DEVELOPER is characterised by a development *velocity* $V_D(t)$, which describes how fast she/he can complete the implementation of a USERSTORY in terms of Story Points per day. Each DEVELOPER starts the PROJECT with randomly different *initialVelocity* $V_D\ |_{init}$, whose mean is given by $V_{team}(0)/N_{devs}$ [4]. The *initialVelocity* $V_D\ |_{init}$ increases with her/his *skill* level, as shown in equation 5.5.

$$V_D(t_i) = V_D\ |_{init} + log[Skill(t_i)] \tag{5.5}$$

[3] 1 year = 200 working days $\Rightarrow$ 5 years = 1000 days.

[4] $V_{team}(0)$ is the initial team velocity without PP (see sec. 5.5) and N_{devs} is the number of developers of the team.

Finally, the attribute *isAvailable* is a Boolean value that takes into account the current availability of the DEVELOPER to work on a new DEVELOPMENTSESSION. The value of this attribute is normally "true" except when she/he is involved in a DEVELOPMENTSESSION.

Actor: Customer

This Actor represents the customer – or a proxy customer – who pay for the system to be implemented by the Team. She/he indicates the functionalities that the final system must have, in the forms of User Stories (*writeUserStory()*). In addition, she/he chooses the order with which these User Stories have to be implemented (*assignPriorities()*). This assignment is performed randomly choosing one of the three possible values: *Must*, *Should* and *Could*. The probabilities to choose each of these values is given as an input model parameter (priorities probability array, see Tab. A.3).

5.2.3 Model Activities

Each Activity in the model is characterized by, at least, a *startTime* and a *duration*, and has two related events: ACTIVITYSTART and ACTIVITYEND. Generally, an Activity is created when the specific ACTIVITYSTART event is executed by the SIMULATOR. For example, when a RELEASEPLANNINGSTART event is on the top of the SIMULATOR's *eventsQueue*, it is executed and an instance of RELEASEPLANNING activity is created. Figures 5.3 and 5.4 show an extract of the class diagram regarding the identified model Activities and Events. More precisely, an event of ACTIVITYSTART or ACTIVITYEND has an attribute named *activity*. This attribute is aimed to link the

specific event with the proper activity.

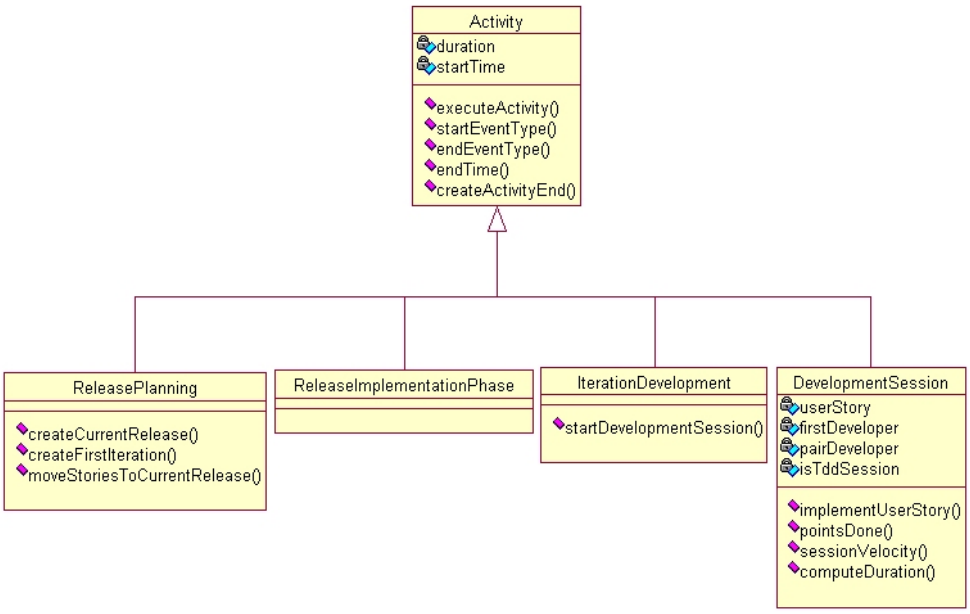


Figure 5.3: *Extract from the class diagram of the identified XP activities.*

The actions performed during the execution of an activity are very different from one kind to another. Let us see in more details each particular case.

Activity: ReleasePlanning

The objective of this activity is to create a plan for the UserStories that have to be implemented in the next Release.

The *duration* of this kind of activity is an input parameter of the Simulator (T_{plan}). In fact, each team can use more or less time for the planning activities. Therefore, it is useful to change this particular parameter value (see Tab. A.2).

Since no particular events occur between the start and end of this activity,

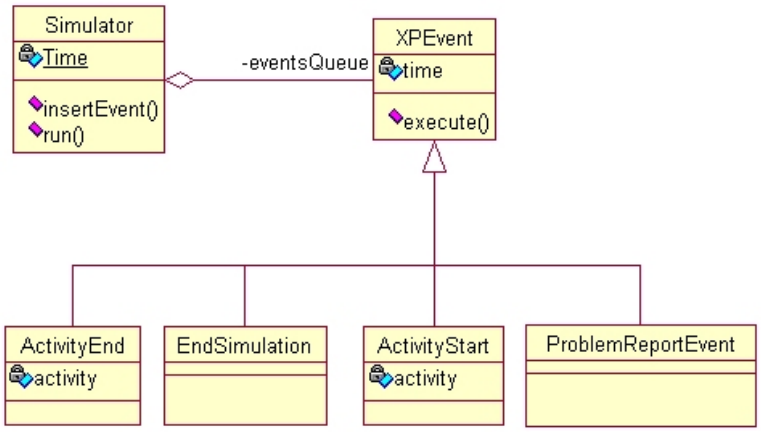


Figure 5.4: *Extract from the class diagram of the identified XP events.*

the end time can be determined. Then, a ReleasePlanningEnd event is inserted into the *eventsQueue* when the ReleasePlanningStart event is executed.

The execution of a ReleasePlanning activity is triggered by the ReleasePlanningEnd event. First of all, the Customer assign the priorities to all the UserStories on the Project container (see sec. 5.2.1). Then, the Team estimates all these UserStories (see sec. 5.2.2).

After that, if a US estimation exceeds a certain limit (a fraction of the week's *capacity*) it will be split into two or more USs. More in details, I assumed that a UserStory is considered to be a "Macro UserStory" if its estimation is greater than half the week's capacity.

The next step consists in choosing the USs which will be implemented in the next Release and consequently assigned to a specific Iteration. The USs are ordered by *priority* (in order: *Must*, *Should* and *Could*), then by *status* (in order: *InProgress*, *ToDo* and *Done*) and, finally, in order of creation.

Once the USs are ordered, a number of them can be moved to the next RELEASE container. Each RELEASE has its own *capacity*, which is given by the actual daily TEAM *velocity* (V_{team}) multiplied by the duration of the RELEASE under schedule (Equation 5.6). The RELEASE is loaded of USs up to its *capacity*, that is the sum of the estimated points of the selected USs doesn't have to exceed the RELEASE *capacity*.

$$capacity = V_{team} \cdot duration \qquad [story\,points] \tag{5.6}$$

If a RELEASE is not full yet, but the next US in the ordered queue is too big, then this US will be split in two USs in such a way that one of these derived USs can be put in the next RELEASE. The other derived US is put in the PROJECT container and will be planned in the next RELEASEPLANNING activity.

Activity: ReleaseImplementationPhase

This activity represents the phase of development of a particular RELEASE. It can be seen as a macro activity, in that it act as a container of other sub-activities.

The RELEASEIMPLEMENTATIONPHASE starts (RELEASEIMPLEMENTATIONPHASESTART event) at the time the RELEASE itself starts, and ends (RELEASEIMPLEMENTATIONPHASEEND) at the end of the last ITERATION planned for the current RELEASE.

At the end of this activity, the associated *endEvent* (ReleaseImplementationPhaseEnd) move the remaining not implemented USs to the *Project*'s stories container, so that they can be planned again in next RELEASEs. In addition, a PROBLEMREPORTEVENT is created and inserted in the SIMU-

LATOR's *eventsQueue*. This particular event will probably create instances of PROBLEMREPORTSTORYS

Actually, this activity doesn't do nothing special and could be removed. The class RELEASE could substitute its role. However, this class has been created to keep a coherent structure in the activity/event model.

Activity: IterationDevelopment

This activity is aimed to manage the development activities that happens during an ITERATION. It is responsible to start new DEVELOPMENTSESSIONS and, if the time scheduled for the current ITERATION is running out, close itself and create a start event for the next ITERATION or RELEASE.

This is the algorithm used. At first, it verify if the remaining time to the planned ITERATION ending is sufficient[5] to start a new DEVELOPMENTSESSION. If so, a new DEVELOPMENTSESSION is created whether there is at least an available DEVELOPER and an USERSTORY inside the ITERATION's *stories* container. Otherwise, an ITERATIONDEVELOPMENTEND event is created, whose execution creates another ITERATIONDEVELOPMENTSTART or, in the case no other ITERATIONS were planned for the current RELEASE, a RELEASEIMPLEMENTATIONPHASEEND event.

If an US is still *InProgress* when an ITERATIONDEVELOPMENT ends, it will be split in two USs. The one partially implemented will be left in the ITERATION with the status *NotCompleted*, while the other one will be moved to the current RELEASE with the status *InProgress*, in order to be completed in the next ITERATION. This new US has an attribute named "*splittedFrom*"

[5]On my hypothesis, the smallest DEVELOPMENTSESSION can lasts 0.1 days.

to take track of the original US from which it was born.

Activity: DevelopmentSession

During a DevelopmentSession a Developer implements the code for a specific UserStory. The duration Δt_{sess} of a session is drawn from a discrete distribution function and can take five different values expressed in working day units: $\Delta t_{sess} = \{0.1, 0.2, 0.3, 0.4, 0.5\}$. Assuming an 8-hour working day, the duration can range from 48 minutes to 4 hours. The probability that a specific session will have a certain duration can be chosen as an input parameter (P_{sess}, see Tab. A.2), in that each team usually has its own rules and habits.

When a DevelopmentSession is created by the IterationDevelopment activity, an associated DevelopmentSessionStart event is created with a given *startTime* and enqueued in the Simulator's events list.

A DevelopmentSession has a Boolean attribute (*isTddSession*) stating whether it will be performed using TDD or not. This attribute is chosen randomly with a probability given by the *TddAdoption* input parameter ($A_{\%}^{Tdd}$). This value can be set up in input in order to take into account of the influence of this practice on the project outputs.

In an XP project the development session would normally be carried out by two programmers working in pairs at a single computer (Pair Programming). However, this practice is rarely adopted in toto. For this reason, the model has an input parameter called *Pair Programming Adoption* ($A_{\%}^{PP}$) that indicates the percent usage of this practice. This parameter expresses the probability that a DevelopmentSession will be performed by two pro-

grammers rather than one.
When a DevelopmentSession is created and assigned to a Developer (*firstDeveloper*), if another Developer is available at that moment, the session will be performed in pair with a probability given by the *Pair Programming Adoption* level.

Once a DevelopmentSessionStart event is executed, the activity *endTime* is calculated, based on the actual status of the US to implement and the moment in the day. In particular, a first random value is chosen and it is possibly reduced to fit in the day.
To avoid the situation that the involved Developers would remain occupied doing nothing when the implementation of a US is finished before the planned session *duration*, the *duration* is possibly reduced based on the points still to implement for the current US.

Now the end session event is created with the appropriate time and enqueued in the Simulator's events list. When executed, it updates all the resources involved in that session (developers, US and the associated source code) and put the US in the current Iteration. After that, it asks the IterationDevelopment activity to plan other sessions or close the Iteration.

Pair Programming influence Development Session performed in pair programming is more efficient than a "solo-programming session" in terms of time required to implement a single UserStory and defects injected [13]. Moreover, the gained Developer's *skill* increases 8% faster when pair programming is applied [59].

I assumed that the *velocity* of a pair of Developers is given by their

average *velocity* increased by 40% ($w_V^{PP} = 0.4$), as reported in some empirical studies [13], [47] (see equation 5.7).

Test Driven Development influence The more complete and accurate experimental study [24] showed that TDD DEVELOPERs took 16% more time than non-TDD DEVELOPERs to develop the same small project. This corresponds to a decrease in the overall development speed of 14%, so I assumed that the *velocity* of a single DEVELOPMENTSESSION, performed following TDD, decreases by the same percentage ($w_V^{Tdd} = -0.14$).

$$V_{sess} = \begin{cases} V_{D1} \cdot (1 + w_V^{Tdd}) & \text{no PP, with TDD} \\ \frac{(V_{D1}+V_{D2})}{2} \cdot (1 + w_V^{Tdd}) \cdot (1 + w_V^{PP}) & \text{with PP and TDD} \end{cases} \tag{5.7}$$

Source code production At the end of a DEVELOPMENTSESSION, new code is generated and the existing code is modified, inevitably injecting a certain number of defects. The level of these changes is affected by stochastic variables influenced by both DEVELOPERs' attributes (*experience* and *skill*) and usage levels of individual XP practices (*TDD* and *Pair Programming*). The produced code is measured through product metrics (such as number of classes, methods, LOCs and defect density).

Equation 5.8 dictates the production of the source code ΔS during a DEVELOPMENTSESSION. It depends on the specific session duration $\Delta t_{sess}(t_j)$, in terms of days, and on the productivity coefficient $P_i(t_j)$.

$$\Delta S_i(t_j) = \Delta t_{sess}(t_j) \cdot P_i(t_j) \qquad i = C, M, L \tag{5.8}$$

The subscript $i = C, M, L$ represents Classes, Methods and LOCs respectively. The productivity coefficient $P_i(t_j)$ is measured in terms of size over

effort [19].

For example, P_L is expressed in terms of LOCs per day and is taken from a log-normal distribution calibrated using product data from the first iterations of the specific project, or from previous projects implemented by the same team. Then, at the end of the j^{th} DevelopmentSession, of duration $\Delta t_{sess}(t_j)$, the number of lines of code associated to the UserStory under development will be increased by $\Delta S_L(t_j) = \Delta t_{sess}(t_j) \cdot P_L(t_j)$. Similarly, a number of $\Delta S_C(t_j)$ classes and $\Delta S_M(t_j)$ methods will be added to the same UserStory.

The Refactoring Model As the system increases in size, its complexity consequently increase. This phenomenon lead to an increase of the project entropy and renders difficult to improve and maintain the system. Refactoring is a technique that allow to reduce and control this inevitably software quality worsening [22].

In this simulation model a very simple Refactoring activity is implemented. With a certain chance p_r, a DevelopmentSession is aimed to refactor the system. In the case it occur, that session will reduce the size of the software code (LOCs, methods and classes related to the UserStory under development) by the same quantities $\Delta S_i(t_j)$ reported in Eq. 5.8. No further defects are injected into the system.

The probability $p_r(t_i)$ that a refactoring session occurs, depends on how much time $\Delta t_r(t_i)$ is passed from the last refactoring session (see Eq. 5.9). This is based on the observation that the system has more needs of being refactored if much time is passed from the last refactoring.

$$p_r(t_i) = P_{r,max} \cdot (1 - e^{-\frac{\Delta t_r(t_i)}{\tau_r}}) \tag{5.9}$$

where $P_{r,max}$ is the maximum value that $p_r(t_i)$ can reach (input model parameter), and τ_r is the time constant of the equation. Every time a refactoring session finishes, the variable $\Delta t_{refact}(t_i)$ is reset to zero.

Furthermore, based on the assumption that the greater is the system size, the more frequent it needs to be refactored, the probability $p_r(t_i)$ also depends on the system size in terms of lines of code (S_L) (see Fig. 5.5). More precisely, $\tau_r = \tau_r(S_L)$, in fact $1/\tau_r(S_L)$ can be seen as the speed at which $p_r(t_i)$ raises to $P_{r,max}$ (Eq. 5.10).

$$\tau_r(S_L(t_i)) = \tau_{r,max} \cdot e^{-\frac{S_L(t_i)}{K_{refact}}} \tag{5.10}$$

where $\tau_{r,max}$ and K_{refact} are two coefficients used to calibrate the model.

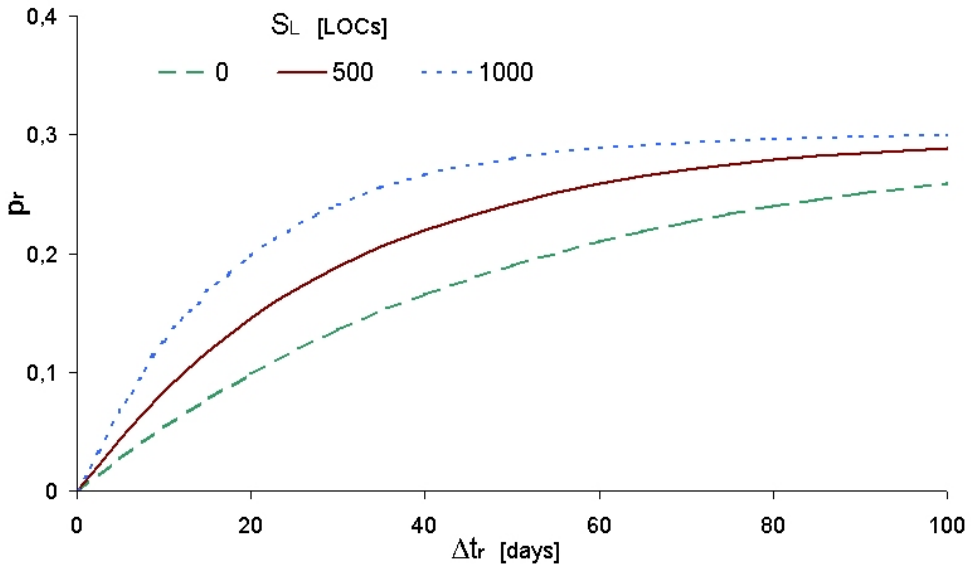


Figure 5.5: *Refactoring probability p_r plotted for three values of the system size S_L.*

The defects injection model The number of defects $\Delta D(t_j)$ injected after the j^{th} development session is given by the following Equation 5.11:

$$\Delta D(t_j) = \Delta S_L(t_j) \cdot d(t_j) \tag{5.11}$$

where $\Delta S_L(t_j)$ is the number of lines of code produced during that DevelopmentSession and $d(t_j)$, expressed in defects per lines of code, is the so called "defect injection rate".

As reported in [6], [30], [57], typical values for defect injection rate are 25 defects/KLOC for the design phase and 15 defects/KLOC for the coding phase. These values normally vary depending on Developer *skills*.

This phenomenon has been studied in depth in the Devnani-Chulani model [16], where it was found that the number of defects injected varies with programmers capability (*skill* in this model), other than with the specific development phase. Tab. 5.1 shows the ratings for the different levels of programmer capability.

Table 5.1: *Defect injection rate multipliers (Devnani-Chulani model).*

Capability	Design	Coding
Very High	0.85	0.76
Nominal	1.00	1.00
Very Low	1.17	1.32

The model incorporates these results in the following way: the number of defects injected per KLOC during a DevelopmentSession decreases linearly with Developer *skill* level[6]. A "Very High" capability corresponds

[6]For a pair programming session, I considered the more skilled programmer.

to the maximum skill level ($S_{max} = 10$), while "Very Low" corresponds to a minimum skill level ($S_{min} = 1$). In addition, since in an XP process design and coding are not performed in different phases, but concurrently during the same DevelopmentSession, the defect injection rates for the two different phases have been combined. The resulting rate ranges from $d_{min} = (1.17 \cdot 25 + 1.32 \cdot 15)$ to $d_{max} = (0.85 \cdot 25 + 0.76 \cdot 15)$ when passing from the minimum to the maximum *skill* level (see table 5.1).

Other studies have found that pair programming produces better quality projects than solo programming (see section 3.2 for details). I introduced this behaviour into the model as follows: in a pair programming session the defect injection rate will be that of the more skilled Developer of the pair, further reduced by 15% ($w_d^{PP} = -0.15$). This percentage may well be greater in that empirical studies have found that the number of functional tests decreased by 15%, therefore the number of defects would be probably more smaller.

It has also been found in [64] and [66] that when using the TDD practice the number of detected defects decreases by about 40% compared to projects developed using a more traditional approach (code then test). Based on these findings, I decreased the number of defects introduced during a session by a further 40% when TDD is adopted ($w_d^{Tdd} = -0.4$).

5.3 The Simulator Engine

The Simulator engine is composed of a set of other classes which enable the model to be simulated.

Simulator The simulator itself. It contains an instance variable "*eventsQueue*" which contains the list of events that have to be executed,

ordered according to their *time* attribute. The core routine is "*run*". It implements the following algorithm:

```
Class: Simulator
Method: run
Algorithm:
   while: (eventsQueue is not empty)
        do: {
            currentEvent := first event of eventsQueue.
            remove the first event of eventsQueue.
            if (currentEvent is the end simulation event)
                then: {
                    Time := time of currentEvent.
                    terminate simulation.}
                else: {
                    execute currentEvent.
                    Time := time of currentEvent}
```

Parameters This class contains all the parameters used by the system (see Appendix A).

XPPracticeInfluence Contains the specific parameters related to the XP practices in the model and implements some of their influence to the model behavior.

Random1 Implements the stochastic functions used in the model.

5.4 A Quick View to the Simulation Flow

After having chosen and customized the input project and model parameters (see Appendix A), a project simulation is created and initialized with a number N_{devs} of DEVELOPERS, and an initial number N_{US} of USs which identify the project main requirements and represent a preliminary evaluation of the project size.

Once a PROJECT has been properly created, a first RELEASEPLANNINGSTART event, with $time = 1$, is put in the SIMULATOR's *eventsQueue*. Another event is created and enqueued: this is an ENDSIMULATION event, a special event aimed to stop a simulation run when executed. It is set up with $time = T_{max}$ which is an input parameter used with the objective to close a project after a specific time. After this value T_{max}, a project is considered as failed. This is useful when a budget limit has been chosen for a specific project.

Now the simulation can start. The first activity to be executed is the RELEASEPLANNING. During this activity, the USs in the PROJECT container are prioritised by the CUSTOMER and estimated by the TEAM. If an US estimation exceeds a certain limit it will be split into two or more USs (Section 5.2.3).

Once the USs have been ordered by *priority*, a number of them is chosen to be implemented in the next RELEASE (Section 5.2.3). Then, a new RELEASE and its first ITERATION start.

Inside an ITERATION, design and development of the scheduled USs are performed during DEVELOPMENTSESSION activities. The time actually spent to implement each US is affected by the estimation error and by the

velocity of the developers who have worked on it. In addition, it is also influenced by percent usage of *Pair Programming* and *Test Driven Development (TDD)* practices.

In some cases not all the planned USs are completed within one ITERATION. These USs are replanned and implemented in subsequent ITERATIONS.

At the end of each ITERATION, a new ITERATION or RELEASEPLANNING activity is created. It depends on how many ITERATIONS have been planned for the current RELEASE.

Upon completion of each RELEASE the TEAM reports any problems encountered in the system developed so far. The number of problems is proportional to system defect density in terms of residual defects per US. These reports are planned by the TEAM like the other USs (PROBLEMREPORTSTORY (PR-US)) each of which has an associated US affected by the problem found (*storyToDebug*). The implementation of each PR-US has the only effect of reducing the number of defects of the related US.

5.5 A Better Explanation of the Practices' Usage Level

This section gives an explanation of how the variation of the usage levels of *Pair Programming* and *Test Driven Development (TDD)* practices has been implemented and influence the model entities. It is fundamental to understand this mechanism in order to appropriately use this feature of the simulator.

Actually, for each practice p there are two input parameters regarding

the usage level: the "*initial practice usage level*" ($A^p_{\%}|_{init}$) and the "*practice usage level*" ($A^p_{\%}$). The first is the real level of adoption of the particular practice in the context of the project used to collect the data needed to set the input parameters for the simulator. In other words, it is a data that characterize the real team we are going to simulate.
The second parameter is the desired adoption level. The simulations are performed using this second value. Thanks to it, the simulator user can simulate how the project will be performed varying the adoption level of that particular practice.

Currently, the "*initial practice usage level*" $A^p_{\%}(t_0)$ is used only for one purpose: to determine the DEVELOPERS' *initialVelocity* attribute when they are created. In fact, as described in section 5.2.2, each DEVELOPER starts the PROJECT with randomly different *initialVelocity* $V_D\ |_{init}$, whose mean is given by $V_{team}(t_0)/Size_{team}$. The input parameter $V_{team}(t_0)$ is sufficient to describe the TEAM as a whole, but insufficient when we want to characterize each individual DEVELOPER. In fact, $V_{team}(t_0)/Size_{team}$ correctly define the average individual development speed whether that TEAM were not using Pair Programming. In the case this practice were used – at a specified usage level $A^{PP}_{\%}(t_0)$ – the proper TEAM velocity ($V_{team}(0)$, without pair programming) needs to be determined (Eq. 5.12).

$$V_{team}(0) = V_{team}(t_0) \cdot \frac{1 + A^{PP}_{\%}(t_0)}{1 + A^{PP}_{\%}(t_0) \cdot w^{PP}(V_{team})} \tag{5.12}$$

Having found $V_{team}(0)$, the average individual DEVELOPER *velocity* can be determined as $V_{team}(0)/N_{devs}$ (see sec. 5.2.2).

XP practices influence some variables in the model. For each variable ν, the level of influence depends on the adoption percentage $A^p_{\%}$ of each particular practice p and on a coefficient w^p_ν, as reported in Eq. 5.13.

$$\nu(A^p_{\%}) = \nu(0) \cdot (1 + A^p_{\%} \cdot w^p_\nu) \tag{5.13}$$

A generic variable ν can be affected by more than one practice. For this reason, instead of a single value $A^p_{\%}$, we have a vector $\overline{A_{\%}}$ of adoption level values. Similarly, instead of w^p_ν we have $\overline{w_\nu}$. Then, the Eq. 5.13 becomes a vectorial equation, as reported in Eq. 5.14.

$$\nu(\overline{A_{\%}}) = \nu(\overline{0}) \cdot (1 + A^{PP}_{\%} \cdot w^{PP}_\nu) \cdot (1 + A^{Tdd}_{\%} \cdot w^{Tdd}_\nu) \ldots \tag{5.14}$$

$\nu(\overline{A_{\%}})$: resulting variable affected by the various practices;

$\nu(\overline{0})$: initial value of the variable under consideration when the usage level of all the practices is set to 0%;

$A^p_{\%}$: usage level for the specific practice p. A value of 0 stays for "practice not adopted" (0%), while 1 "practice fully adopted" (100%);

w^p_ν : specific weight regulating the impact of the practice p on the variable ν.

An application example of equation 5.14 is shown by equation 5.7. In that case $A^p_{\%}$ can be only 0 or 1 because it is referred to a single DevelopmentSession that can be performed in pair or not and in TDD or not (see sec. 5.2.3).
Another use of this influence model is reported in page 59 where it is explained how TDD and PP affect the defect injection rate.

Through this practice influence mechanism it is simple to improve the simulation model adding other practices that affect the model variables. Also, it is simple to adjust the weight a single practice has on a specific variable ν varying the coefficients w_ν^p (see Tab. A.4).

5.6 Calibration and Validation

One of the major problems in process simulation is the effective calibration and validation of the simulator developed. In order to achieve this, data sets gathered from real projects are required. However, these data are difficult to obtain for several reasons. The greater part of real projects are developed by privately-owned companies that, for obvious reasons, are generally reluctant to publish data regarding their internal development process.

Also, it is difficult to find companies that develop software using XP and systematically collect information about their development process. Moreover, should this be so, there is no guarantee that the degree of detail of the information gathered is sufficient to perform proper calibration and validation of simulation models. I cite two XP projects where tracking activity has been conducted systematically and where the data available are sufficiently detailed: *Repo Margining System* [33] and *Market Info* [7], [8].

In order to calibrate the parameters of the simulation model, I used some input variables (see Tab. 5.2) drawn from the *Repo Margining System* project [33], such as number of developers, release duration and so on. Also, I used the project and process data gathered during the first two iterations. Then, I simulated the evolution of the whole project.

A number of simulation runs were performed using these input param-

Table 5.2: *Input parameters taken from the Repo Margining project used to calibrate the model. A story point [pts] corresponds to 30 minutes of work.*

Input parameters		Value
Number of initial USs		27
Number of developers		4
Mean (stand. dev.) of USs estimation	[pts]	15 (12)
Mean (stand. dev.) of USs estimation error		2 (0.9)
Initial Team velocity	[pts/day]	16
Number of Iterations per Release		3
Typical iteration duration	[days]	10
Typical session duration	[days]	0.2
Mean (stand. dev.) of Locs productivity	[Locs/day]	156 (80)
Mean (stand. dev.) of Methods productivity	[Methods/day]	7 (3)
Mean (stand. dev.) of Classes productivity	[Classes/day]	1.8 (0.3)

Table 5.3: *Comparison between simulation results averaged over 200 runs and the Repo Margining System project. Standard deviations are reported in parenthesis. A story point [pts] corresponds to 30 minutes of work.*

Output variable		Simulation	Real Project
Days	[days]	61.9 (13.5)	60
User Stories		28.2 (1.6)	29
Total Estimated Effort	[pts]	525.6 (132.3)	474
Total Actual Effort	[pts]	829.7 (203.7)	793
Number of Releases		2.5 (0.6)	2
Iterations per Release		2.7 (0.3)	3
Lines of Code	[KLOC]	9.9 (1.5)	9.8
Methods		440.8 (71.3)	454
Classes		113.2 (17.6)	107

Table 5.4: *Input parameters taken from the M@rket Info project used to validate the model. A story point [pts] corresponds to 30 minutes of work.*

Input parameters		Value
Number of initial USs		131
Number of developers		6
Mean (standard deviation) of USs estimation	[pts]	18.4 (12)
Mean (standard deviation) of USs estimation error		1.7 (0.7)
Initial Team velocity	[pts/day]	28
Number of Iterations per Release		1
Typical iteration duration	[days]	7.5
Typical session duration	[days]	0.2

eters. I then calibrated iteratively the model parameters so as to obtain a better fit of the final results of the real project. In Tab. 5.3 the simulation outputs are compared with those taken from the *Repo Margining System* case study.

In order to validate the simulation model I used the *Market Info* project changing the project dependent parameters as reported in Tab. 5.4.

I had no information about the estimation error on single user stories completed during the first iterations. I hypothesized that the estimation error decreases according to the specific team's experience. Actually there are no empirical case studies that support this assumption. However, I used this hypothesis to choose the input project parameter. In particular, I was aware that the M@Info team was more experienced than the Repo Margining team. Also, I observed that the average velocity of a single developer of the

Table 5.5: *Comparison between simulation results averaged over 200 runs and the M@rket Info project. Standard deviations are reported in parenthesis. A story point [pts] corresponds to 30 minutes of work.*

Output variable		Simulation	Real Project
Days	[days]	225.3 (29.1)	198
User Stories		158.2 (5.83)	168
Total Actual Effort	[pts]	4160.0 (470.1)	4118

M@Info team was 16% higher than a developer of the Repo Margining team. So, I decided to decrease the average estimation error by the same percentage.

The simulation results obtained for this project are shown in Tab. 5.5. As can be seen, the simulated project data agree fairly well with the real project data, demonstrating that the model provides sufficiently valid and accurate results. Actually, I was not able to validate the output variables related to software production because sufficient information was not available about code productivity during the initial project phases.

Conceptual model validation was also performed interviewing some software engineers familiar with the XP process itself. The proposed approach was presented, and its various concepts – roles, activities and artifacts – were explained in detail. I collected positive feedback on my approach and on the specific parameter values used in the simulation model.

Chapter 6

Experimental Results

6.1 Research Hypotheses

The main objective of this research work was to explore the influence of some key XP practices (*Test-First Programming* and *Pair Programming*) on the evolution of an XP project. I investigated how the outputs of a typical XP project (*Repo Margining System*) varied changing the usage levels of these two practices. I focused on output variables related to effort, size, quality and released functionalities. That is, total working days (*Days*), size of the project in terms of lines of code (*KLOCs*), residual defect density (*Defects/KLOC*) and released User Stories (*User Stories*).
The research hypotheses were as follows:

Hypothesis A: The residual defect density of the project using both PP and TDD is different from that obtained without PP and/or TDD.

Hypothesis B: The number of working days needed to complete the same number of functionalities using both PP and TDD is different from that

Table 6.1: *The four project research conditions.*

	PP	TDD
case 1	0%	0%
case 2	0%	100%
case 3	100%	0%
case 4	100%	100%

without PP and/or TDD.

Hypothesis C: The number of lines of code needed to implement the same number of functionalities using both PP and TDD is different from that without PP and/or TDD.

Hypothesis D: The number of released User Stories using both PP and TDD is equal to that without PP and/or TDD.

6.2 Simulation Results

Keeping the same input parameters reported in Tab. 5.2, I examined the simulation outputs in the four project conditions shown in Tab. 6.1.
For each case 200 simulation runs have been performed. The summarized results are reported in Tab. 6.2.

The results with PP = 100% and TDD =100% (*case 4*) are identical to those shown in Tab. 5.3, obtained simulating the Repo Margining System project.

Looking at *case 2* (Tab. 6.2), we can see that some outputs for the same simulated project varied only when Pair Programming was not adopted.

Table 6.2: *Comparison of simulation results averaged over 200 runs for four project conditions (see Tab. 6.1). Standard deviations are reported in parenthesis. Detailed results in Appendix B.*

Output variable	*case 1*	*case 2*	*case 3*	*case 4*
Days	41.8 (11.9)	48.8 (11.3)	53.0 (10.8)	61.9 (13.5)
User Stories	28.2 (1.5)	28.4 (1.6)	28.2 (1.5)	28.2 (1.6)
Defects/KLOC	65.0 (4.0)	40.2 (2.4)	51.7 (3.3)	31.4 (1.8)
KLOCs	11.8 (2.6)	13.4 (2.4)	9.0 (1.4)	9.9 (1.5)

I found that by not using Pair Programming at all, the duration of the project (in terms of working days) decreases by 21%. The number of User Stories remains fairly similar in both cases. On the other hand, defect density increases by 28% when PP is not practised. In addition, the number of LOC is increased by 35% by not using PP.

Consequently, we can say that the use of Pair Programming increases the total development cost (working days), but this is offset by better quality project (in terms of defect density) and by a better design, in terms of fewer lines of code per User Story (0.47 KLOC/US against 0.35 KLOC/US). These results agree fairly well with those reported in [13] and [47], as previously described in section 3.2.

What happens when the TDD practice is not adopted? Starting from the Repo Margining results (*case 4* - Tab. 6.2), I proceeded without using TDD (*case 3* - Tab. 6.2). As can be observed, not using TDD decreases the duration of the project by 14%, residual defect density increases by 65% and the number of released USs is fairly similar, while there is a 9% reduction in

project size.

The extra time taken by TDD may be attributed to the time needed to develop test cases. In addition, test cases lead to an increment of the project's source instructions, as confirmed by the simulation results. On the other hand, notwithstanding of the minor advantage that less effort is needed to complete the project, project quality deteriorates significantly. This is due to the fact that the model does not include the quality assurance and rework phases, typical of a classic approach (testing after coding, then rework).

Let us point out that all these experiments has been performed setting the number of problems found by the team (problem reports rate K_{PR}) to zero, which is the model's only mechanism for simulating a sort of pre-release system testing phase.

In the extreme case I varied the usage of both practices simultaneously from TDD=PP=100% (*case 4*) to TDD=PP=0% (*case 1*). I obtained a 32% reduction in time and a 19% size increase, while the defect density doubled.

In order to validate the research hypotheses, I performed a series of statistical tests: two-sided t-test for normal samples and KS-test for non-normal samples[1]. As can be observed from the summarized results reported in Tab. 6.3, my hypotheses (A, B, C and D) were confirmed with a statistical significance of 95%, except for one case. In fact, when I exclude the use of Pair Programming, the defect density does not differ significantly, though we can observe from Tab. 6.2 a 26% increase between the average values.

Finally, I tried to adjust the problem reports rate coefficient K_{PR} so as to

[1]The Kolmogorov-Smirnov test (KS-test) tries to determine if two datasets differ significantly. Basically, it is used in the place of the t-test when the datasets are non-normally distributed [56].

Table 6.3: *Results of the two-sided tests ($\alpha = 0.05$) obtained for the four hypotheses under test.*

	case 4 vs case 1		*case 4 vs case 2*		*case 4 vs case 3*	
	H_0	$P-value$	H_0	$P-value$	H_0	$P-value$
Hyp_A (t-test)	rejected	$2.7 \cdot 10^{-66}$	accepted	0.57	rejected	$8.0 \cdot 10^{-67}$
Hyp_B (ks-test)	rejected	$1.9 \cdot 10^{-39}$	rejected	$9.0 \cdot 10^{-18}$	rejected	$2.6 \cdot 10^{-9}$
Hyp_C (t-test)	rejected	$5.0 \cdot 10^{-18}$	rejected	$3.4 \cdot 10^{-52}$	rejected	$4.3 \cdot 10^{-9}$
Hyp_D (ks-test)	accepted	1.0	accepted	0.9	accepted	0.9

determine the additional cost required to achieve the same quality (in terms of residual defects) of the system developed using both TDD and PP (see Tab. 6.4).

As can be observed when neither practice is adopted an extra cost of 145% is incurred in terms of days needed to complete the project with the same quality level (*case 1* and *case 4* - Tab. 6.4). That is, only 28% (41.8 days) of the total effort (151.9 days) was spent developing system functionalities, while the remaining (72%) was devoted to improving project quality.

I observed similar behaviour omitting the use of only one the two XP practices, but on a smaller scale. When Pair Programming was omitted (*case 2*), the team had to spend an additional 24% of the time to achieve the same quality as the project developed using both TDD and PP (*case 4*). Here, the effort devoted to quality assurance and rework (36%) is lower than the previous case, because the use of TDD made it possible to achieve higher quality during the implementation phase.

The last case (*case 3*) shows an increase of 78% in the final time and

Table 6.4: *Comparison of simulation results averaged over 200 runs for four project conditions, varying the problem reports rate K_{PR}. Standard deviations are reported in parenthesis.*

Output variable	*case 1* $K_{PR} = 0.07$	*case 2* $K_{PR} = 0.04$	*case 3* $K_{PR} = 0.07$	*case 4* $K_{PR} = 0.00$
Days	151.9 (28.1)	76.8 (26.5)	110.1 (27.1)	61.9 (13.5)
User Stories	29.5 (2.4)	28.6 (2.1)	29.4 (2.2)	28.2 (1.6)
Defects/KLOC	**31.6 (5.6)**	**30.8 (3.4)**	**31.1 (2.9)**	**31.4 (1.8)**
KLOCs	12.56 (2.4)	13.7 (3.3)	9.4 (1.4)	9.9 (1.5)

the extra time spent on improving project quality accounts for 52% of the total effort. A similar situation was studied in [49] and [31], where Pair Programming is compared to a quality assurance process but with discordant conclusions.

6.2.1 Further Analysis on the Simulation Model

In order to better understand the mechanism influencing the output variables of the simulation model, I analyzed in depth its variations against some key input parameters. In particular, the usage levels of TDD and PP has been varied and it has been observed how the four output variables under study consequently changed. This analysis has been performed simulating a typical project of 30 initial User Stories and a team made up of 6 developers having an initial team velocity of 25 points/day. All the other parameters have been kept equals to those reported in Tab. 5.2. Fig. 6.1 reports the simulation results obtained gradually changing the usage level of TDD ($A_{\%}^{Tdd}$) from 0%

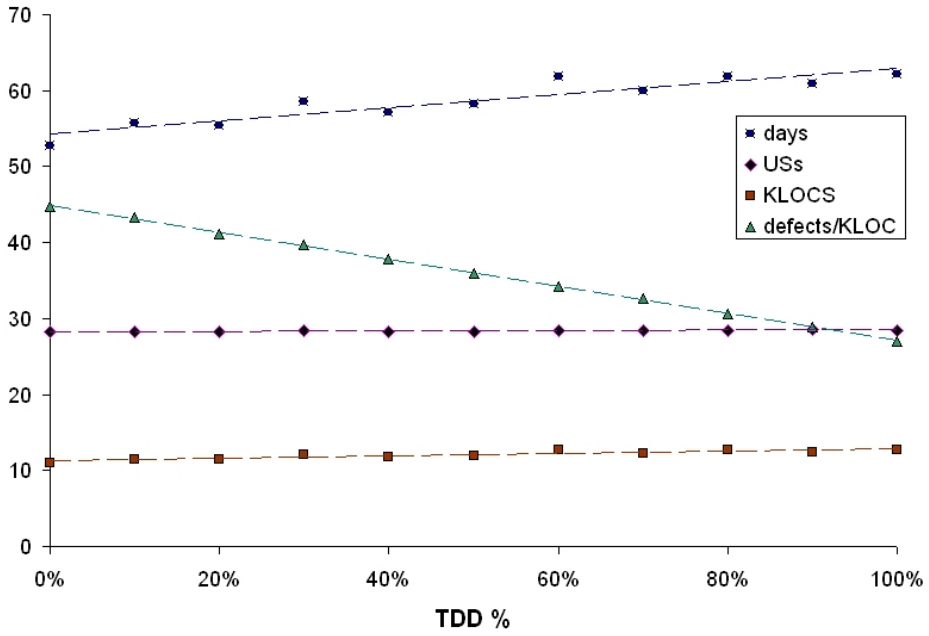


Figure 6.1: *Average results, over 200 simulation runs, obtained varying the TDD usage level.*

to 100%, keeping constant all the other parameters. Using the sensitivity function reported in Eq. 2.1 and linearly approximating the variation of the output variables, we have obtained the following sensitivity results: the more sensitive variable is the residual defect density (*Defects/KLOC*), having obtained a sensitivity value of $S_{Tdd}^{defects} = -25\%$. A sensitivity value of 7% has been obtained by both *KLOCs* and *Days* variables, while the last variable (*USs*) is unsensitive to the variations of this parameter.

Therefore we can state that the adoption of the adoption of TDD has a large positive impact on the quality that can be obtained for a specific project, in that it lead to a reduction of the system defectiveness, and slightly increase the system size and the project duration.

In Fig. 6.2 has been reported the results of an analogous analysis per-

formed varying the PP adoption level parameter ($A^{PP}_{\%}$). In this particu-

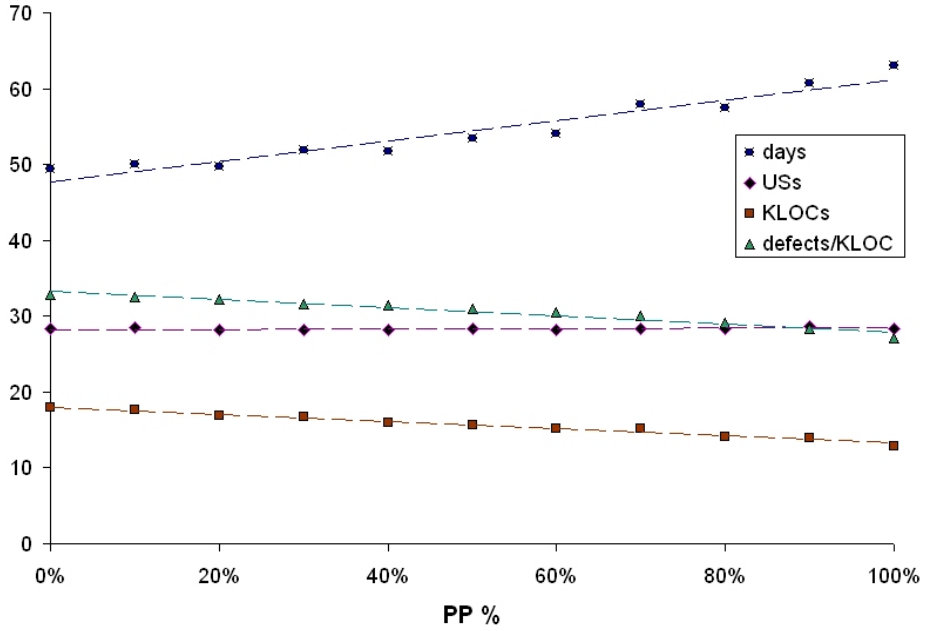


Figure 6.2: *Average results, over 200 simulation runs, obtained varying the PP usage level.*

lar case, the following sensitivity values have been obtained: $S^{days}_{PP} = 12\%$, $S^{defects}_{PP} = -9\%$, $S^{klocs}_{PP} = -15\%$, $S^{USs}_{PP} = 0\%$. Therefore, based on these simulation model results, we can say that Pair Programming adoption significantly reduces the number of defects of a software system. Also, it negatively affects the system size and, on the other hand, positively affect the project duration, increasing the working days needed to complete the same number of *User Stories*.

Special attention have to be given to understand the difference between the desired PP adoption level ($A^{PP}_{\%}$) and the actual use of this practice by the team. In fact, as described in section 5.2.3, $A^{PP}_{\%}$ expresses the probability that a DevelopmentSession will be performed by two programmers rather

than one. However, a DevelopmentSession can be performed in pair only if two Developers are both available at the same time. For this reason not all the planned pair sessions will be actually performed by two Developers. Therefore, the actual percentage of pair programming adoption will be less than $A^{PP}_{\%}$, as shown in Fig. 6.3.

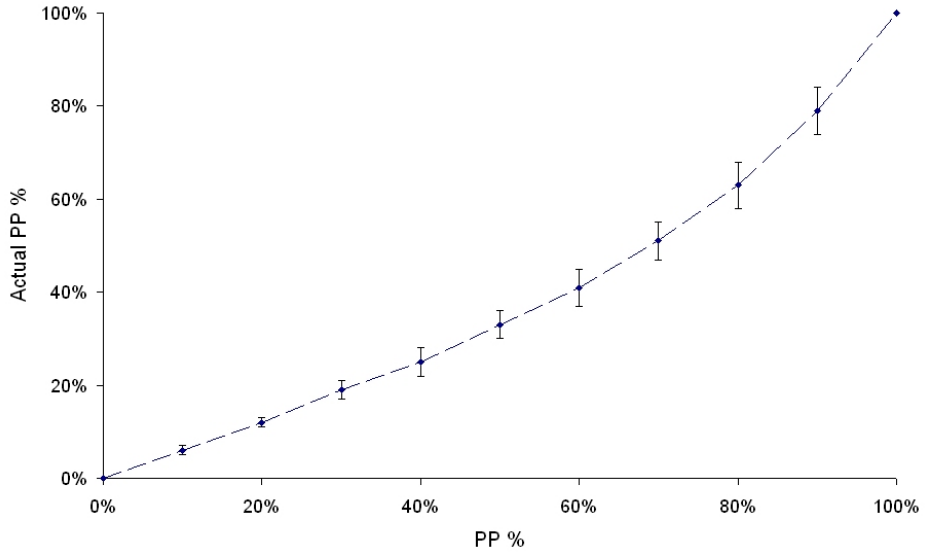


Figure 6.3: *Average results, over 200 simulation runs, of the actual percentage of PP sessions against the desired PP adoption level.*

Since the actual adoption percentage of PP is always different from the desired adoption percentage $A^{PP}_{\%}$, the sensitivity results have to be computed again: $S^{days}_{PP} = 9\%$, $S^{defects}_{PP} = -6\%$, $S^{klocs}_{PP} = -10\%$, $S^{USs}_{PP} = 0\%$. As can be seen from these results and from Fig. 6.4, the output variables are less sensitive to the variation of the actual PP level rather than the desired one. However, similar trends have been obtained although in a smaller scale.

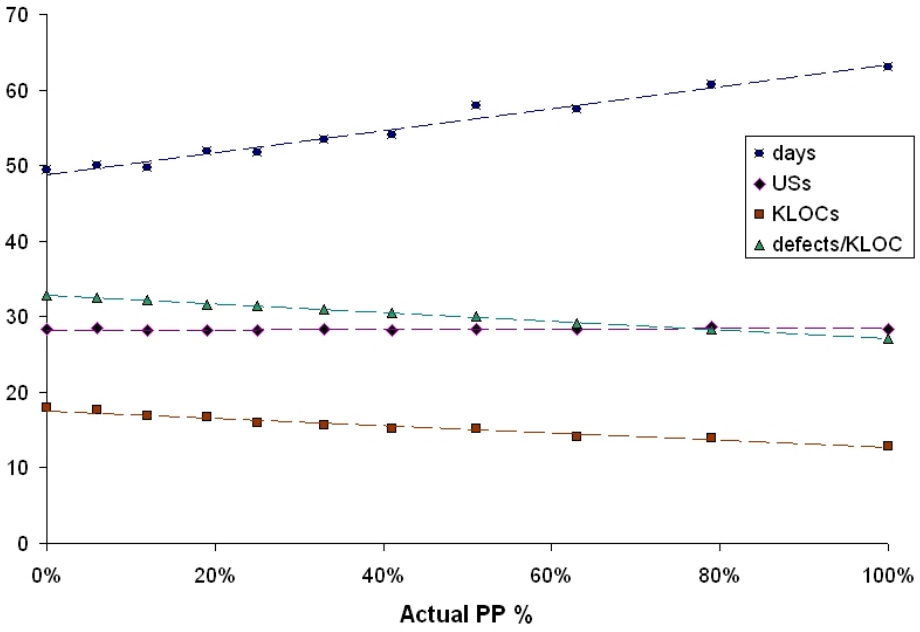
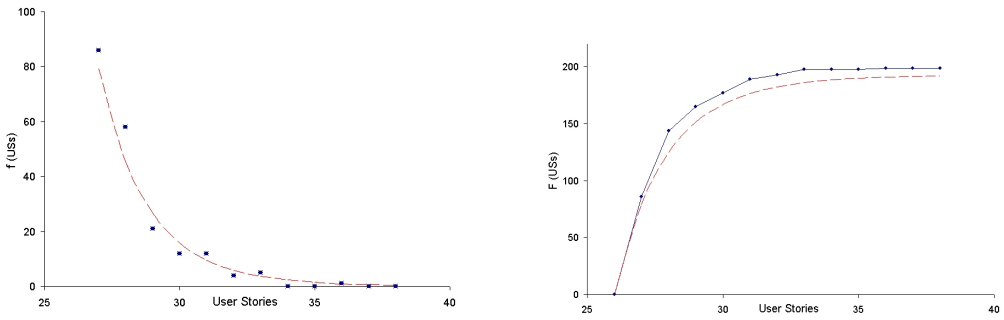


Figure 6.4: *Average results, over 200 simulation runs, obtained varying the PP usage level.*

6.2.2 Statistical Analysis of the Results

Concerning the statistical nature of the results, a set of Kolmogorov-Smirnov test [56] has been performed to determine the probability distributions that better fit the output data. In particular, I focused my attention on the four variables analysed in the study obtained simulating the RepoMargining project (Tab. 6.1, *case 4*): total working days (*Days*), size of the project in terms of lines of code (*KLOCs*), residual defect density (*Defects/KLOC*) and released User Stories (*User Stories*).

The test results – reported in Tab. 6.5, with a 95% confidence level – showed that *Defects/KLOC* and *KLOCs* can be represented as normal distributions, while *Days* is better fit by a log-normal distribution. The last variable (*User Stories*) is better described by a power law distribution as shown in Fig. 6.5.



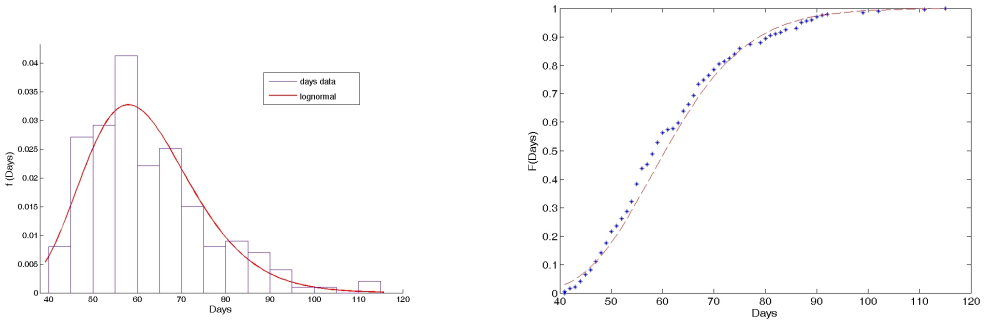
(a) *Frequency* (b) *CDF*

Figure 6.5: *The User Story output variable is better described by a power law distribution.*

The box plots in Fig. 6.9 provide a visual representation of the statistical nature of the results obtained simulating the Repo Margining System project (*case 4*). Descriptive statistic of these results is reported in Tab. B.4.

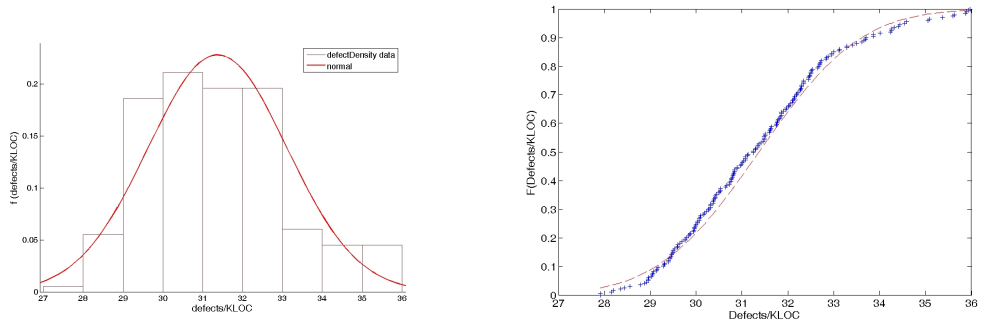
Table 6.5: *Results of the two-sample KS-tests ($\alpha = 0.05$) for the best fit distribution of the output variables.*

Variable	Distribution	Equation	Parameters		H_0	$P-value$
Days	Log-normal	$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln(x)-\mu)^2}{2\sigma^2}}$	$\mu = 4.10$	$\sigma = 0.21$	accepted	0.11
User Stories	Power Law	$f(x) = k \cdot x^a$	$k = e^{54.82}$	$a = -15.31$	accepted	0.43
Defects/KLOC	Normal	$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$	$\mu = 31.37$	$\sigma = 1.75$	accepted	0.54
KLOCs	Normal	$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$	$\mu = 9.85$	$\sigma = 1.53$	accepted	0.10



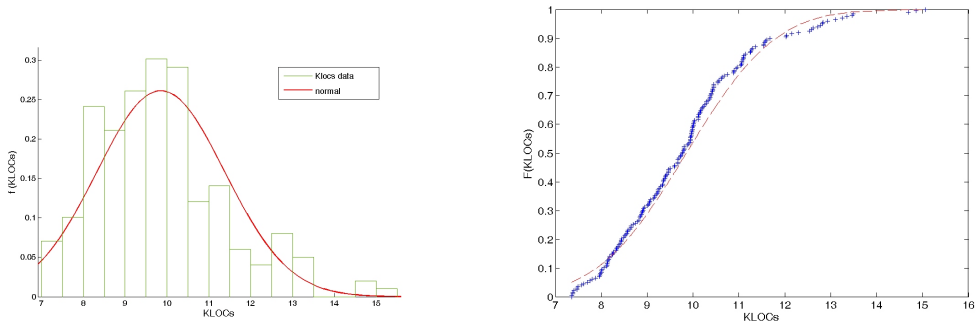
(a) *PDF* (b) *CDF*

Figure 6.6: *Days output variable distribution and best fit with a log-normal distribution.*



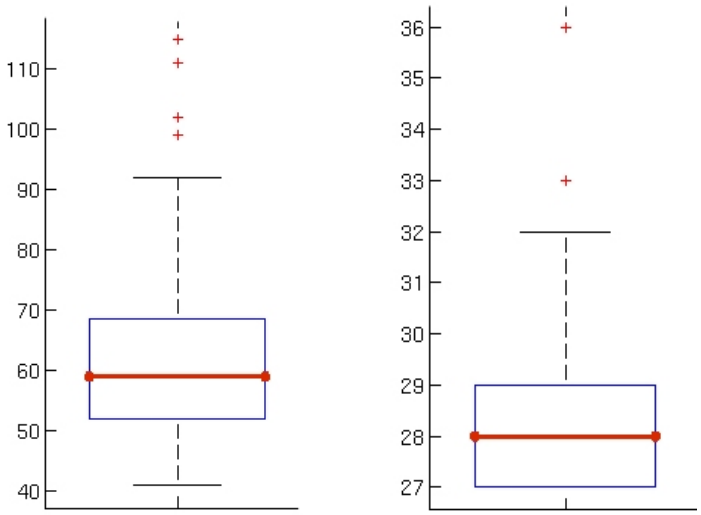
(a) *PDF* (b) *CDF*

Figure 6.7: *Defects/KLOC output variable distribution and best fit with a normal distribution.*

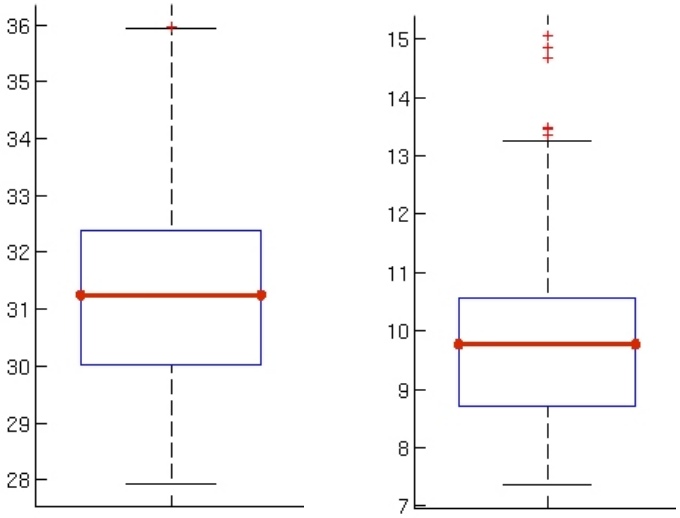


(a) *PDF* (b) *CDF*

Figure 6.8: *KLOCs output variable distribution and best fit with a normal distribution.*



(a) *Days* (b) *User Stories*



(c) *Defects/KLOC* (d) *KLOCs*

Figure 6.9: *Box plots for the four output variables under consideration (case 4).*

Chapter 7

Conclusions and Future Work

An Extreme Programming (XP) process simulation model has been developed and here presented for assessing the effectiveness of some XP practices and for obtaining a deeper understanding of its dynamics.

To gain a greater insight into the XP process it was chosen of developing the whole simulation model in our research group following an XP-like methodology. In this way, an increased knowledge of the process under study has been gained, diminishing the gap between theory and practice.

The use of the XP process allowed us to develop the simulation model in an incremental way. The model started with an initial embryonic version and then evolved, step by step, to its current status. Calibration and validation were also performed in an incremental manner. This provided useful feedbacks, suggesting how to adjust the equations, parameters and model events in order to obtain more realistic results.

In addition, the use of XP has led to a software implementation with a simple and flexible structure, provided with a fairly complete test suite. This will enable anyone to improve the simulation model and add further features,

such as the implementation of other XP practices, in a more confident and easier way.

The model has been calibrated and validated using data obtained from two different real projects. Then, the variation in usage level of two key XP practices – *Pair Programming* and *Test First Programming* – has been simulated. It has been found that increasing the usage of such practices significantly diminishes product defectiveness. That is, the adoption of both these practices greatly improved the software quality and reliability. On the other hand, the results showed that greater effort, in terms of working days, was required to implement the same number of functionalities.

More in details, the adoption of *Test First* has shown a large positive impact on the quality that can be obtained for a specific project, in that it led to a reduction of the software defectiveness, slightly increasing the system size and the project duration. At the same way, *Pair Programming* adoption significantly reduces the number of defects and slightly affect the project duration, increasing the working days needed to complete the same *User Stories*. However, it negatively affects the system size, decreasing the total number of lines of code.

In order to estimate the cost of the adoption of the two practices, a quality assurance process was introduced into the model. Then, it has been calibrated in such a way that similar levels of project quality can be obtained in the various cases. In this way, it can be evaluated the extra cost needed to reach same final residual defectiveness when one of the two practices are not used.

The results showed that, when *Pair Programming* was omitted, the team

had to spend an additional 24% of the time to achieve the same quality as the project developed using both the practices. A greater increase (78%) in the final time has been observed when *Test First* was not adopted. In the worst case, when neither of the practices were used, it has been found an extra cost of 145% in terms of days needed to complete the same number of functionalities.

I would point out that the developed model is not a complete representation of the intrinsic complexity of these practices and of the development process itself. A model is always a strong simplification of the real world. This simulation model can be used in support of managers and software engineers to "qualitatively" estimate and forecast the behaviour of a certain project when changes on the development process are introduced.

Least but not last, it can be used in support of researchers in that, trying to model an Extreme Programming process using the empirical findings reported to date, it has enlightened some issues that warrant further investigation. Estimation errors varying team experience and skill, the influence of *Pair Programming* on developers' learning curve, how code productivity is influenced by *Pair Programming* and *Test First*, are some of the hypotheses made because no consistent data were available. To improve model reliability and obtain more realistic results requires the creation of a more comprehensive knowledge base on agile software development.

Bibliography

[1] Tarek Abdel-Hamid and Stuart E. Madnick. *Software Project Dynamics: an Integrated Approach*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[2] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[3] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

[4] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change- Second Edition*. Addison-Wesley, 2004.

[5] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[6] Barry W. Boehm. Developing small-scale application software products: Some experiment results. In *IFIP Congress*, pages 321–326, 1980.

[7] PierGiuliano Bossi. Extreme programming applied: a case in the private banking domain. In *Proceedings of OOP2003*, 2003.

[8] Piergiuliano Bossi. Using actual time: learning how to estimate. In *XP2003 Conference Proceedings*, pages 244–253, 2003.

[9] John Brewer and Jera Design. Extreme programming faq. Url: http://www.jera.com/techinfo/xpfaq.html, 2000.

[10] Christopher Browne. Christopher browne's web pages - computer languages: Smalltalk. Url: http://www.ntlug.org/ cbbrowne/smalltalk.html, 2000.

[11] CampSmalltalk. Sunit. Url: http://sunit.sourceforge.net/, 2000.

[12] Lan Cao. A modeling dynamics of agile software development. In *Companion of 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 46–47. ACM Press, 2004.

[13] Alistair Cockburn and Laurie Williams. The costs and benefits of pair programming. In *Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2000)*, Cagliari, Sardinia, Italy, June 2000.

[14] Donald C. Craig. Extensible hierarchical object-oriented logic simulation with an adaptable graphical user interface. Master of science, School of Graduate Studies, Department of Computer Science, Memorial University of Newfoundland, 1996.

[15] L. Crispen and L. House. *Testing Extreme Programming*. Addison Wesley, ma: addison wesley pearson education edition, 2003.

[16] Sunita Devnani-Chulani. Results of delphi for the defect introduction model. Technical Report USC-CSE-97-504, Computer Science Department, University of Southern California, Los Angeles, 1997.

[17] Jerry Drobka, David Noftz, and Rekha Raghu. Piloting xp on four mission-critical projects. *IEEE Softw.*, 21(6):70–75, 2004.

[18] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226–237, March 2005.

[19] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: a rigorous and pratical approach*. PWS Publishing Company, 1996.

[20] George S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer Series in Operations Research and Financial Engineering. Springer-Verlag, Berlin, 2001.

[21] J. W. Forrester. *Industrial Dynamics*. Cambridge MA: Productivity Press, 1961.

[22] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[23] Martin Fowler. The new methodology. Published on: http://martinfowler.com/articles/newMethodology.html, 2003.

[24] Boby George and Laurie Williams. An initial investigation of test driven development in industry. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 1135–1139. ACM Press, 2003.

[25] Boby George and Laurie Williams. An initial investigation of test driven development in industry. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 1135–1139. ACM Press, 2003.

[26] Paul Hodgetts and Denise Phillips. Extreme adoption experiences of a B2B start-up. Published on: www.extremejava.com/files/eXtremeAdoptioneXperiencesofaB2BStartUp.pdf, 2001.

[27] Hanna Hulkko and Pekka Abrahamsson. A multiple case study on the impact of pair programming on product quality. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 495–504, New York, NY, USA, 2005. ACM Press.

[28] The System Dynamics in Education Project MIT. Road maps: A guide to learning system dynamics. Published on: http://sysdyn.clexchange.org/road-maps/home.html, 2000.

[29] PRandall W. Jensen. Management impact on software cost and schedule. Published on: http://www.stsc.hill.af.mil/crosstalk/1996/07/manageme.asp, 1996.

[30] T. Capers Jones. Measuring programming quality and productivity. *IBM Systems Journal*, 17(1):39–63, 1978.

[31] Gerold Keefer. Pair programming: An alternative to reviews and inspections? *Cutter IT Journal*, 18(1), 2005.

[32] Marc I. Kellner, Raymond J. Madachy, and David M. Raffo. Software process simulation modeling: Why? What? How? *The Journal of Systems and Software*, 46(2–3):91–105, April 1999.

[33] KlondikeTeam. Tracking-aworkingexperience. Published on: http://www.communications.xplabs.com/paper2001-2.html, 2000.

[34] Verna Knapp. The smalltalk simulation environment. In *WSC '86: Proceedings of the 18th conference on Winter simulation*, pages 125–128, New York, NY, USA, 1986. ACM Press.

[35] Wolfgang Kreutzer. *System Simulation - Programming Styles and Languages*. Addison Wesley, Reading (U.S.A.), 1986.

[36] S. Kuppuswami, K. Vivekanandan, Prakash Ramaswamy, and Paul Rodrigues. The effects of individual xp practices on software development effort. *SIGSOFT Softw. Eng. Notes*, 28(6):6–6, 2003.

[37] S. Kuppuswami, K. Vivekanandan, and Paul Rodrigues. A system dynamics simulation model to find the effects of xp on cost of change curve. In *XP2003 Conference Proceedings*, pages 54–62, 2003.

[38] Kim Man Lui and Keith C. C. Chan. When does a pair outperform two individuals? In *XP2003 Conference Proceedings*, pages 225–233, 2003.

[39] Kim Man Lui and Keith C.C. Chan. Test driven development and software process improvement in china. In Jutta Eckstein and Hubert Baumeister, editors, *XP2004 Conference Proceedings*, volume 3092 of *LNCS*, pages 219–222. Springer, 2004.

[40] Michele Marchesi. The new xp. Published on: http://www.agilexp.org, 2005.

[41] Robert H. Martin and David Raffo. A model of the software development process using both continuous and discrete models. *Software Process: Improvement and Practice*, 5(2-3):147–157, 2000.

[42] Frank Maurer and Sebastien Martel. Extreme programming: Rapid development for web-based applications. *IEEE Internet Computing*, 6(1):86–90, 2002.

[43] Frank Maurer and Sebastien Martel. On the productivity of agile software practices: An industrial case study. Published on: http://sern.ucalgary.ca/ milos/Library.htm, 2002.

[44] Vojislav B. Misic, Hudson Gevaert, and Michael Rennie. Extreme dynamics: Modeling the extreme programming software development process. In *Proceedings of ProSim04 workshop on Software Process Simulation and Modeling*, pages 237–242, 2004.

[45] Matthias M. Müller and Oliver Hagner. Experiment about test-first programming. *IEEE Proceedings - Software*, 149(5):131–136, 2002.

[46] Jerzy Nawrocki and Adam Wojciechowski. Experimental evaluation of pair programming. In *12th European Software Control and Metrics Conference (ESCOM 2001)*, 2001.

[47] John T. Nosek. The case for collaborative programming. *Commun. ACM*, 41(3):105–108, 1998.

[48] Peter Ball University of Strathclyde. Introduction to discrete event simulation. Published on: http://www.dmem.strath.ac.uk/ pball/simulation/simulate.html, 2001.

[49] Frank Padberg and Matthias Muller. Analyzing the cost and benefit of pair programming. In *Proceedings of Ninth International Software Metrics Symposium*, pages 166–177, 2003.

[50] Charles J. Poole and Jan Willem Huisman. Using extreme programming in a maintenance environment. *IEEE Software*, 18(6):42–50, 2001.

[51] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[52] Armin Rohrl and Stefan Schmiedl. Interview to Alistair Cockburn on programming productivity. Url: http://www.approximity.com/produktiver_programmieren/cockburn_en.html, 2002.

[53] Armin Rohrl and Stefan Schmiedl. Interview to Ron Jeffries on programming productivity. Url: http://www.approximity.com/produktiver_programmieren/jeffries_en.html, 2002.

[54] Ioana Rus, Michael Halling, and Stefan Biffl. Supporting decision-making in software engineering with process simulation and empirical studies. *International Journal of Software Engineering and Knowledge Engineering*, 13(5):531–545, 2003.

[55] Thomas J. Schriber and Daniel T. Brunner. Inside discrete-event simulation software: how it works and why it matters. In *WSC '99: Proceed-*

ings of the 31st conference on Winter simulation, pages 72–80. ACM Press, 1999.

[56] David Sheskin. *The Handbook of Parametric and Nonparametric Statistical Procedures*. CRC Press, 2003.

[57] T. Thayer, M. Lipow, and E. Nelson. *Software Reliability*. North Holland, 1978.

[58] Onur M. Ulgen and Timothy Thomasma. Simulation modeling in an object-oriented environment using smalltalk-80. In *WSC '86: Proceedings of the 18th conference on Winter simulation*, pages 474–484, New York, NY, USA, 1986. ACM Press.

[59] K. Vivekanandan. *The Effects of Extreme Programming on Productivity, Cost of Change and Learning Efficiency*. PhD thesis, Department of Computer Science, Ramanaujam School of Mathematics and Computer Sciences, Pondicherry University, India., 2004.

[60] Wayne W. Wakeland, Robert H. Martin, and David Raffo. Using design of experiments, sensitivity analysis, and hybrid simulation to evaluate changes to a software development process: a case study. *Software Process: Improvement and Practice*, 9(2):107–119, 2004.

[61] Don Wells. What is extreme programming? Published on: http://www.extremeprogramming.org/what.html, 1999.

[62] Laurie Williams and Robert Kessler. *Pair Programming Illuminated*, chapter 4: Overcoming Management Resistance to Pair Programming, pages 33–44. Addison-Wesley Longman Publishing Co., Inc., 2002.

[63] Laurie Williams, Robert R. Kessler, Cunningham Cunningham, and Ron Jeffries. Strengthening the case for pair programming. *IEEE Software*, 17(3):19–25, July/August 2000.

[64] Laurie Williams, E.Michael Maximilier, and Madlen Vouk. Test-driven development as a defect-reduction practice. In *Proceedings of the 14th Symposium on Software Reliability Engineering (ISSRE'03)*, pages 34–45, 2003.

[65] William A. Wood and William L. Kleb. Exploring XP for Scientific Research. *IEEE Software*, 20(3):30–36, 2003.

[66] Randy A. Ynchausti. Integrating unit testing into a software development teams process. Published on: http://www.agilealliance.org/articles/ynchaustirandyaintegr/file, 2001.

[67] George W. Zobrist and James V. Leonard. *Object Oriented Simulation: Reusabilty, Adaptability, Maintainability.* IEEE Press, 1997.

Appendix A

Parameters

Table A.1: *Input parameters required to start a simulation and main output variables of the simulation model.*

Input parameters	Main output variables
Number of initial USs	Released User Stories
Number of developers	Defect density [defects/KLOC]
Mean and stand. dev. of USs estimation [pts]	Classes
Mean and stand. dev. of USs estimation error	Methods
Initial Team Velocity [pts/day]	Lines of code
Number of Iterations per Release	Working days
Typical iteration duration [days]	Total Estimated Effort [pts]
Typical session duration [days]	Total Actual Effort [pts]
Productivity coefficients (P_C, P_M, P_L)	

Table A.2: *Project specific parameters.*

PARAMETER	DESCRIPTION	DEFAULT VALUE
N_{devs}	Number of DEVELOPERs of the TEAM.	
N_{US}	Number of initial USs, which identify the project main requirements and represent a preliminary evaluation of the project size.	
T_{max}	Maximum duration for a project can be considered to be failed. After this time the simulation run will be stopped (*maxProjectDuration*).	150 days
$V_{team}(t_0)$	Initial TEAM *velocity*.	
$A^p_{\%}(t_0)$	Typical historical usage level of the specific practice p for your team.	100%
$A^p_{\%}$	Simulation usage level for the specific practice p.	100%
T_{plan}	Typical time spent in planning activities for each release.	1 day
P_{sess}	Session duration probability array: each element gives the probability of choosing one of the following durations for a development session expressed in working day units: $\Delta t_{sess} = \{0.1; 0.2; 0.3; 0.4; 0.5\}$.	$[0; 1; 0; 0; 0]$
P_C, P_M, P_L	Productivity coefficients for Classes, Methods and LOCs respectively, measured in terms of size over effort [Classes/day, Methods/day, LOCs/day]	1.8; 7.0; 156.0

Table A.3: *Inner model parameters.*

Parameter	Description	Default Value
K_{macro}	Coefficient used to identify a Macro UserStory. It indicates the minimum number of USs that should be implemented during a week	2
P_{prior}	Priorities probability array: each element gives the probability of choosing one of the three priority values: *Must*, *Should* and *Could*.	$[\frac{1}{3}; \frac{1}{3}; \frac{1}{3}]$
K_{learn}	Learning coefficient (see Eq. 5.4). A different value of this coefficient is used when a DevelopmentSession is performed in pair programming (within parentheses).	0.009 (0.00972)
K_{PR}	Problem reports rate: coefficient used to set the number of problem reports created before the end of each Release.	0.00
$P_{r,max}$	Maximum value for the probability that a refactoring session will occur.	0.3
$\tau_{r,max}$	Maximum refactoring τ: is the initial value assumed by τ, when the system size $S_L = 0$, in the refactoring model (see Eq. 5.10).	50 [days]
K_{refact}	Coefficient used to calibrate the refactoring model (see Eq. 5.10).	1000 [LOCs]
d_{min}	Minimum defect injection rate: minimum number of defects injected per KLOC.	32.65 [defects/KLOC]
d_{max}	Maximum defect injection rate: maximum number of defects injected per KLOC.	49.05 [defects/KLOC]
$P_{fix}\|^{min}$	Probability to fix a defect when TDD is not used at all.	0.73
$P_{fix}\|^{max}$	Probability to fix a defect when TDD is fully used.	0.97

Table A.4: *Influence weights of XP practices on model variables.*

Practice	TDD		PP	
Variable	*weight*	*value*	*weight*	*value*
V_{sess}	w_V^{Tdd}	-0.14	w_V^{PP}	0.40
d_{inj}	w_d^{Tdd}	-0.40	w_d^{PP}	-0.15

Appendix B

Detailed Results

Table B.1: *Descriptive statistics for the output variables of case 1.*

Variable	*Mean*	*Std. Dev.*	*Min*	*Max*	*Mode*	*Median*
Days	41.76	11.86	141.00	24.00	41.00	40.00
User Stories	28.24	1.45	36.00	27.00	27.00	28.00
Defects/KLOC	65.04	4.02	76.49	53.83	N.A.	65.14
KLOCs	11.77	2.57	34.15	7.37	12.07	11.49

Table B.2: *Descriptive statistics for the output variables of case 2.*

Variable	*Mean*	*Std. Dev.*	*Min*	*Max*	*Mode*	*Median*
Days	48.78	11.27	86.00	29.00	43.00	46.00
User Stories	28.35	1.56	34.00	27.00	27.00	28.00
Defects/KLOC	40.18	2.40	48.43	34.33	N.A.	40.34
KLOCs	13.38	2.35	23.23	9.07	11.32	13.23

Table B.3: *Descriptive statistics for the output variables of case 3.*

Variable	*Mean*	*Std. Dev.*	*Min*	*Max*	*Mode*	*Median*
Days	52.96	10.80	94.00	31.00	54.00	52.00
User Stories	28.24	1.50	35.00	27.00	27.00	28.00
Defects/KLOC	40.18	2.40	48.43	34.33	N.A.	40.34
KLOCs	8.96	1.42	15.53	6.36	8.63	8.77

Table B.4: *Descriptive statistics for the output variables of case 4.*

Variable	*Mean*	*Std. Dev.*	*Min*	*Max*	*Mode*	*Median*
Days	61.86	13.54	41.00	115.00	55.00	59.00
User Stories	28.22	1.60	27.00	36.00	27.00	28.00
Defects/KLOC	31.37	1.75	27.92	35.97	N.A.	31.25
KLOCs	9.85	1.53	9.67	7.36	15.07	9.78

Table B.5: *Descriptive statistics for the output variables of case 1 with* $K_{PR} = 0.07$.

Variable	*Mean*	*Std. Dev.*	*Min*	*Max*	*Mode*	*Median*
Days	151.89	28.05	192.00	79.00	178.00	164.00
User Stories	29.54	2.42	39.00	26.00	27.00	29.00
Defects/KLOC	31.62	5.57	52.50	22.34	N.A.	30.60
KLOCs	12.55	2.44	19.23	7.77	11.11	12.35

Table B.6: *Descriptive statistics for the output variables of case 2 with* $K_{PR} = 0.038$.

Variable	*Mean*	*Std. Dev.*	*Min*	*Max*	*Mode*	*Median*
Days	76.78	26.51	187.00	29.00	55.00	73.00
User Stories	28.61	2.11	38.00	27.00	28.00	28.00
Defects/KLOC	30.75	3.39	39.93	23.67	27.05	30.59
KLOCs	13.72	3.26	37.61	8.74	16.05	13.12

Table B.7: *Descriptive statistics for the output variables of case 3 with* $K_{PR} = 0.07$.

Variable	*Mean*	*Std. Dev.*	*Min*	*Max*	*Mode*	*Median*
Days	110.08	27.07	194.00	56.00	94.00	108.00
User Stories	29.43	2.18	36.00	27.00	28.00	29.00
Defects/KLOC	31.06	2.94	38.70	23.85	N.A.	30.99
KLOCs	9.37	1.40	13.03	6.21	9.51	9.26