TOWARDS A PHILOSOPHICAL UNDERSTANDING OF AGILE SOFTWARE METHODOLOGIES:

THE CASE OF KUHN VERSUS POPPER

Mandy Northover

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF COMPUTER SCIENCE
IN THE FACULTY OF
ENGINEERING, BUILT ENVIRONMENT AND INFORMATION TECHNOLOGY
UNIVERSITY OF PRETORIA
SOUTH AFRICA

JUNE 2008

Abstract

Title:	*Towards a Philosophical Understanding of Agile Software Methodologies: The Case of Kuhn versus Popper*
Author:	Mandy Northover
Supervisor:	Professor Derrick Kourie
First Co-Supervisor:	Doctor Andrew Boake
Second Co-Supervisor:	Doctor Stefan Gruner
Department:	Computer Science
University:	University of Pretoria
Degree:	Master of Computer Science

This dissertation is original in using the contrasting ideas of two leading 20^{th} century philosophers of science, Karl Popper and Thomas Kuhn, to provide a philosophical understanding, firstly, of the shift from traditional software methodologies to the so-called Agile methodologies, and, secondly, of the values, principles and practices underlying the most prominent of the Agile methodologies, *Extreme Programming (XP)*. This dissertation will take a revisionist approach, following Fuller—the founder of social epistemology—in reading Popper against Kuhn's epistemological hegemony.

The investigations in this dissertation relate to two main branches of philosophy—epistemology and ethics. The epistemological part of this dissertation compares both Kuhn and Popper's alternative ideas of the development of scientific knowledge to the Agile methodologists' ideas of the development of software, in order to assess the extent to which Agile software development resembles a scientific discipline. The investigations relating to ethics in this dissertation transfer concepts from social engineering—in particular, Popper's distinction between piecemeal and utopian social engineering—to software engineering, in

order to assess both the democratic and authoritarian aspects of Agile software development and management.

The use of Kuhn's ideas of scientific revolutions and paradigm shift by several leading figures of the Agile software methodologies—most notably, Kent Beck, the leader of the most prominent Agile software methodology, Extreme Programming (XP)—to predict a fundamental shift from traditional to Agile software methodologies, is critically assessed in this dissertation. A systematic investigation into whether Kuhn's theory as a whole, can provide an adequate account of the day-to-day practice of Agile software development is also provided.

As an alternative to the use of Kuhn's ideas, the critical rationalist philosophy of Karl Popper is investigated. On the one hand, this dissertation assesses whether the epistemological aspects of Popper's philosophy—especially his notions of falsificationism, evolutionary epistemology, and three worlds metaphysics—provide a suitable framework for understanding the philosophical basis of everyday Agile software development. On the other hand, the aspects of Popper's philosophy relating to ethics, which provide an ideal for scientific practice in an open society, are investigated in order to determine whether they coincide with the avowedly democratic values of Agile software methodologies.

The investigations in this dissertation led to the following conclusions. Firstly, Kuhn's ideas are useful in predicting the effects of the full-scale adoption of Agile methodologies, and they describe the way in which several leaders of the Agile methodologies promote their methodologies; they do not, however, account for the detailed methodological practice of Agile software development. Secondly, several aspects of Popper's philosophy, were found to be aligned with several aspects of Agile software development. In relation to epistemology, Popper's principle of falsificationism provides a criterion for understanding the rational and scientific basis of several Agile principles and practices, his evolutionary epistemology resembles the iterative-incremental design approach of Agile methodologies, and his three worlds metaphysical model provides an understanding of both the nature of software, and the approach advocated by the Agile methodologists' of creating and sharing knowledge. In relation to ethics, Popper's notion of an open society provides an understanding of the rational and ethical basis of the values underlying Agile software development and management, as well as the piecemeal adoption of Agile software methodologies.

Acknowledgments

Firstly, I would like to express my sincere gratitude to my three supervisors. Professor Derrick Kourie's wealth of experience, paired with his attention to detail, provided the foundation for the supervision of this study. I would like to thank him for his guidance, constructive criticism, and invaluable suggestions for improving this dissertation. I'm indebted to Doctor Andrew Boake, who, during his inspiring Honours lectures, planted the seed in my head, which germinated into this dissertation. His vast industry experience also provided an important perspective to this dissertation. I'd like to thank Doctor Stefan Gruner, who, despite joining the supervisorship towards the end of this study, provided invaluable insights from a broad perspective, especially in relation to the philosophical content of this dissertation. I'd also like to thank Doctor Gruner for the various opportunities he presented, the motivation, and the encouragement. I'm sincerely grateful to all three supervisors for the contributions they have made to both this study, and the papers we have published together.

Secondly, I would like to thank my husband, Alan, without whose guidance, encouragement, patience, and unfailing support, this dissertation would surely not have come to fruition. I'm particularly indebted to him for all the sacrifices he has made on my behalf—not only during this study but also throughout my undergraduate and postgraduate studies. My admiration, respect, and love for him deepens every day.

Thirdly, I would like to thank my immediate family for their loving support, and especially their patience throughout my studies.

Finally, I would like to thank the National Research Foundation as well as the University of Pretoria, who supported this study financially by means of research grants and bursaries.

Table of Contents

List of Figures

Chapter 1

Introduction

The starting point for this study is the long-standing *software crisis*[1], and the contemporary view—first proposed by Brooks (1987)—that there is "no silver bullet" for software engineering. This dissertation will investigate the attempts made by the so-called "lightweight" or "Agile"[2] software methodologists to propose their methodologies as a "silver bullet" to solve the software crisis. In particular, the attempt by Kent Beck—the founder and leader of the most prominent Agile software methodology, Extreme Programming (XP)—will be investigated. While such attempts at a "silver bullet" for the software crisis have been the topic of much research to date, this dissertation will uniquely use the ideas of two leading 20^{th} century philosophers of science, *Karl Popper* and *Thomas Kuhn*, to provide a philosophical understanding of Agile software methodologies, which may reveal the specific reasons these methodologies have been proposed as a "silver bullet" for the software crisis.

Despite several obvious objections to an approach of using the *philosophy of science* to help understand a crisis in the software engineering discipline, there are equally compelling reasons that can justify such an approach, namely:

1. Software engineering is a young discipline and, as such, has not yet been subjected to intense philosophical analysis, as have many other more established disciplines; this

[1] The software crisis started in the mid 1960s when, for the first time in computer history, the cost of software started to exceed the cost of hardware. The frequency of over-budget and over-schedule projects, some of which also caused damage to property and even resulted in the loss of life, increased dramatically. Some believe that the software crisis ended in the mid 1980s but others believe it to still be at the heart of the discipline today.

[2] Agile software methodologies emerged during the early 1990s in response to the challenges of the Internet era, which made it necessary for software methodologies to be flexible in the face of rapid change. The background to these methodologies and their place in the history of software engineering will be described in detail in the following chapter.

dissertation can, therefore, be seen as a starting point for such a philosophical debate;

2. Science currently enjoys a status of epistemological hegemony and most disciplines have, as a result, aspired to scientific status, software engineering being no exception; it seems necessary to understand what this 'scientific status' could mean, since there are competing ideas on what makes a discipline 'scientific';

3. The philosophy of science debate is a well-established and rigorous one, and has been used to illuminate many other disciplines; thus it may well cast some light on the software engineering discipline too;

4. Software engineering, while not a science in the epistemological sense (since it does not attempt to discover laws of nature), is nonetheless considered *scientific*—at least in its methodology—and hence a comparison with the sciences may prove illuminating;

5. Several authors from the software engineering discipline have made use of the theories of various philosophers to help illuminate the theory and practice of their discipline, although these uses have been largely unsystematic; this dissertation aims to use the ideas of Kuhn and Popper systematically and critically to illuminate the theory and practice of Agile software development;

6. Several leading figures of the Agile and XP software methodology movements, most notably Beck, have both explicitly and implicitly used Kuhn's ideas—especially his notion of 'paradigm shift'—in their literature relating to the apparent software crisis;

7. Kuhn's philosophy of science has itself achieved epistemological hegemony in academia, which could be questioned, especially in relation to the way certain Agile software methodologists apparently use his ideas;

8. Few philosophers of science besides Popper have discussed the ethical implications of science (their emphasis being on scientific method); Popper's concept of an *open society* may help to provide a basic framework for assessing both the democratic and authoritarian aspects of Agile software development and management;

9. The use of the insights of the philosophy of science to illuminate software engineering is a novel and bold approach, which may yield valuable conclusions, even if it may later be found to be mistaken.

Over the past four decades, software engineering (SE) has emerged as a discipline in its own right, though it has roots both in classical engineering and in computer science. Since it is a relatively young discipline, its philosophical foundations and premises are not yet well understood. The difficulty of software engineering, as far as philosophical reflections are concerned, is its hybrid character. Software engineering relates itself methodologically to classical engineering, which is based on material sciences. However, it is not equivalent to classical engineering since it does not deal with matter. Software engineering also differs notably from classical engineering, in the frequency of change in the artefact produced. Furthermore, software engineering should not be equated with its parent—computer science—and its parent's parent—mathematics—though traces of both can still be found in the discipline today.

The philosophy of science discipline will be taken as a starting point for the philosophical reflections in this dissertation since this discipline is often used to understand and improve the *theory and practice* of another discipline—in this case, the software engineering discipline. The viability of using classical philosophy of science to illuminate aspects of software engineering may be questioned since software engineering does not primarily aim at knowledge as such, but at the deliberate construction of usable artefacts, to which knowledge is only a means. Knowledge in the software engineering discipline, therefore, seems to be more idiographic than nomothetic[3], which raises the question whether or not the practice of software engineering can justifiably be called a "science", although examples do exist of software engineers who speak, with a peculiar mixture of confidence and optimism, about "software science" as opposed to "software engineering"[4].

Despite all these difficulties in relation to reflecting philosophically on the software engineering discipline, there have recently been attempts by certain software engineers to identify the philosophical foundations of their discipline, using the theories of various professional philosophers. However, these attempts towards a philosophy of software engineering have not only been rather uncritical, but have also been largely incomprehensive and unsystematic, at least in terms of using any single philosophy or school of philosophy to

[3]Software engineers aim to produce useful software artefacts. Therefore, idiographic aspects—those relating to individual artefacts or events—seem to be more relevant to software engineering than nomothetic aspects—those relating to general laws of nature. In the philosophy of science discipline, the inverse is true since philosophers, unlike software engineers, aim to discover general laws of nature. Note that these laws of nature should not to be confused with the so-called laws of human nature.

[4]See, for example, EAS (2008) and SEA (1998).

understand aspects of the software engineering discipline. In particular, there has been no systematic use of any particular philosopher's ideas to understand the philosophical basis of either:

1. the shift in some quarters away from the so-called "traditional" software methodologies towards the more "lightweight" or "Agile" software methodologies; or

2. the values, principles and practices underlying the most prominent of the Agile methodologies, *Extreme Programming (XP)*, whose principle goal it is to be flexible in the face of rapid change.

This dissertation aims to use the contrasting ideas of Popper and Kuhn to provide a philosophical understanding of these two aspects. Throughout this dissertation, XP will be used as the representative when comparisons are made between scientific methodologies and Agile software methodologies[5]. In order to understand XP better, this dissertation will also consider the type of methodology, which Beck defines his methodology in opposition to, namely, the traditional *Waterfall model*.[6]

The contrasting ideas of Popper and Kuhn regarding the nature of science are central to the philosophy of science discipline. Although a debate between the two philosophers organised by Lakatos, a previous student of Popper's and an important philosopher of science in his own right, did not take place in the end, Kuhn's supposedly pluralistic vision of science is generally thought to have triumphed over Popper's apparently more monolithic one. Recently, however, the founder of "social epistemology", *Steve Fuller*, has reopened the *Kuhn vs Popper* debate, arguing that its outcome has been wrongly judged and that, in the process, both parties have been misunderstood. Fuller (2003) takes a revisionist approach, reading Popper's ideas against Kuhn's. This dissertation aims to extend the *Kuhn vs Popper* debate to the software engineering discipline, taking Fuller's revisionist approach in his book as a basis. Fuller argues that Kuhn represents the authoritarian and Popper the libertarian tendencies in science policy:

[5]The English word "methodology" is conceptually closer to the German word "Methode" rather than to the German word "Methodologie": it means a framework of related methods rather than a fully elaborated science or theory of methods.

[6]In this sense, Beck's approach is similar to Popper's negative approach to philosophy: Popper always dissents from or defines his position against the dominant position, whether falsificationism against verificationism (positivism)—in scientific method—or negative utilitarianism against traditional utilitarianism—in ethics.

> Far from defending openness in science, Kuhn represented scientists who tried to maintain their autonomy in the face of Cold War pressures. In contrast, Popper remained a consistent defender of science as 'the open society'. Much more is at stake here than settling a historical score. The future of science itself depends on understanding the philosophical, political and even religious basis of what separated Kuhn and Popper.[7]

Consequently, the anti-authoritarianism in Popper's *critical rationalist* philosophy, will be a recurring theme throughout this dissertation, especially in part III. Fuller also quotes Lakatos who poignantly reiterates the significance of the *Kuhn vs Popper* debate by stating:

> The clash between Popper and Kuhn is not about a mere technical point in epistemology. It concerns our central intellectual values, and has implications not only for theoretical physics but also for the underdeveloped social sciences and even moral and political philosophy.[8]

This dissertation aims to create an awareness in the software engineering discipline of the importance of the *Kuhn vs Popper* debate, and to assess the potential implications the alternative ideas of Kuhn and Popper—which lie at the heart of the debate—could mean for software methodology. In particular, this dissertation aims to assess critically the explicit and implicit use of Kuhnian ideas—in particular, his notions of "paradigm shift", "scientific revolution", and "crisis"—by several leading figures of the Agile and XP software methodologies to predict a fundamental shift from traditional to Agile software methodologies. The use of Kuhnian concepts by these leading software methodologists, is perhaps not completely surprising if one considers the influence that Kuhn's book *The Structure of Scientific Revolutions* (Kuhn, 1970) had in many academic disciplines, being one of the ten most cited academic works for over thirty years[9]. This dissertation will also investigate whether Kuhn's theory of *revolutionary change* in science—which relates exclusively to epistemology—can, as a whole, constitute a background theory for Agile software development.

As a critical alternative to the use of Kuhn's ideas, this dissertation will uniquely and systematically investigate whether Popper's *critical rationalist* philosophy can provide a

[7] Fuller (2003) (Front cover)
[8] Fuller (2003) (p. 9)
[9] Fuller (2003) (p. 1)

unified philosophical framework for understanding Agile software methodologies. In particular, two aspects of Popper's philosophy—the *epistemological* and the *ethical*—will be used to investigate the underlying philosophical basis of the values, principles and practices of the XP software methodology[10]. On the one hand, this dissertation will assess whether the epistemological aspects of Popper's philosophy provide a suitable framework for understanding the rational and scientific basis of everyday XP software development, and, on the other hand, the aspects of Popper's philosophy relating to ethics will be investigated in order to determine whether they provide an understanding of the avowedly democratic and people-centric approach of XP software development and management.

The remainder of this dissertation is structured as follows, into three main parts. After this introductory chapter, Part I (which comprises chapters 2 and 3) will provide the context for the remaining chapters of this dissertation. Chapter 2 will define certain key terms used throughout this dissertation, and will provide an *historical* account of both scientific methodology—from the Ancient Greeks to the present day—and software methodology—over the past four decades. The latter account will attempt to classify the changes that have occurred in software methodology as either *evolutionary* or *revolutionary*, two concepts, which will be related to the respective philosophies of Popper and Kuhn in this dissertation. Chapter 2 will conclude with a description of the activities that typically occur during the life cycle of an XP project, the purpose of which is to anchor the philosophical discussions of this dissertation in a concrete example.

Chapter 3 will provide an account of the work related to the topic of this dissertation, in which other philosophically-minded software engineers have used—either explicitly or implicitly—the ideas of various professional philosophers to illuminate their discipline. The first part of the chapter will provide a general account of the literature, which applies the ideas of a broad range of philosophers to the software engineering discipline—from Plato to Feyerabend. The second part of the chapter will provide a more specific account of the literature, which applies the contrasting ideas of Popper and Kuhn to the software

[10]From the outset, the reader should note that Beck was not aware of Popper's philosophy when he formulated the XP methodology. During an e-mail correspondence with me in 2006 (after both editions of his own book *Extreme Programming Explained* had already been published), Beck acknowledged not being aware of Popper's philosophy. Hence, Popper's work could not have directly influenced the XP methodology. However, Beck was keenly aware of Kuhn's ideas and acknowledged in the same e-mail correspondence that Kuhn influenced his thinking in terms of helping him, firstly, to anticipate how the markets would react to XP, and, secondly, to communicate across paradigms. Moreover, Beck explicitly cites Kuhn's *Structure* in the annotated bibliography of both editions of *Extreme Programming Explained*.

engineering discipline.

The remainder of the dissertation, chapters 4–7, *systematically* use the contrasting ideas of Kuhn and Popper in relation to Agile software methodologies, and are divided into two further parts, *epistemology* and *ethics*.

The *epistemological* part of this dissertation, Part II, comprises chapters 4 and 5. Chapter 4 will investigate whether Kuhn's concept of "paradigm shift", as used by several leading figures of the Agile methodologies—especially those from the Cutter Consortium, for example, Beck and Yourdon—can provide an explanation for the shift in some quarters away from traditional software methodologies towards Agile software methodologies—especially towards XP. The chapter will also provide a detailed analysis of whether Kuhn's ideas, as a whole, can provide a background theory to the practice of XP software development. Chapter 5 will investigate, using Popper's principle of *falsificationism*, whether software engineering resembles a scientific discipline, and to what extent the XP values, principles and practices are aligned with this principle, and with Popper's related ideas of *criticism* and *error elimination*. Secondly, a comparison will be made between Popper's four-stage model of the *evolutionary theory of knowledge*, and XP's *iterative-incremental* approach to software development, in an attempt to illuminate the similarities between a software methodology and a scientific methodology. Chapter 5 will conclude with an investigation into whether Popper's *three worlds metaphysical model*—which emphasises *objective* knowledge—conflicts with the Agile methodologists' stated reliance on *tacit* or *subjective* knowledge, or whether it can, in fact, provide a more balanced understanding of the way in which Agile software methodologists create and share knowledge. Popper's model will also be used in an attempt to understand the *nature* of software, which is conjectured to inhabit his World 3 of objective logical content.

To conclude the epistemological part of this dissertation, a critical reflection will be provided on the findings as well as on the limitations of using the ideas of Kuhn and Popper in relation to Agile software methodologies.

Part III of this dissertation, which deals mainly with *ethics*, consists of chapters 6 and 7. The motivation for dedicating a part of this dissertation to ethics is that no methodology, whether scientific or software, can ever be value-free. Therefore, the choice of methodology will determine the normative orientation of the community. The aim of Part III is to transfer concepts from *social* engineering to *software* engineering, in order to assess both

the *democratic* and *authoritarian* aspects of Agile software development and management. Chapter 6 will investigate the "open" and "closed" tendencies in Agile software development. In particular, Popper's notions of *utopian* and *piecemeal* social engineering will be used to assess the divergent approaches towards software development of traditional and Agile software methodologists. It will also be investigated whether the XP team can be described in terms of Popper's notion of an *open society*. Furthermore, chapter 6 will investigate the parallels between Kuhn's notion of a "paradigm", and the holistic way in which the XP methodology is *promoted*—according to Johnson—in order to assess whether there is a tension between the avowedly democratic values of the XP methodology and the alleged authoritarian manner in which the methodology is promoted.

Chapter 7 will investigate two other important influences on the Agile ethos, namely, the architect, *Christopher Alexander*, and the authors of *Peopleware*, *DeMarco and Lister*. Firstly, the chapter will investigate whether Alexander's approach of *social engineering architecture* resembles Popper's approach of *piecemeal social engineering*. If fundamental resemblances can found, these may partly account for the apparently Popperian spirit of the XP values, principles and practices, due to Alexander's notable influence on the leaders of the XP methodology. Secondly, the ideas of *DeMarco and Lister* in *Peopleware*—a landmark book in software management which demonstrates that the major problems of software development are human, not technical—regarding *management style* will be investigated, in order to determine whether they have parallels in Popper's advocacy of an open society, and in Popper's critique of *utopian social engineering*.

Chapter 8 concludes this dissertation by reflecting on the findings that have been made throughout, and by drawing several conclusions from them. Appendices A–H follow the conclusion and contain, respectively, a definition of the *Manifesto of Agile Software Development*, a list of the *XP Values, Principles and Practices*, a pictorial representation of the interaction between Popper's three worlds in his *Three Worlds Metaphysical Model*, a brief account of the *History of Computer Ethics*, the definition of the *Software Engineering Code of Ethics and Professional Practice*, an investigation into the similarities between *Free Open Source Software and Popper's Open Society*, *The Open Source Definition*, and *The Free Software Definition*.

Part I

Context

Chapter 2

Historical Background and Definitions

In order to provide a historical background for the discussions throughout this dissertation, this chapter will outline the history of scientific and software engineering methodology. It will also define the key terms that will be used throughout this dissertation. This chapter will conclude with a description of the typical activities that occur during Extreme Programming (XP) software development, the aim of which is to provide an important overview of several concepts specific to XP, many of which will be studied in detail later in this dissertation.

2.1 Definitions

Each of the terms that will be defined in this section have varied definitions in the literature. The aim of the definitions given below is to inform the reader of how these terms are to be understood in the context of this dissertation.

The professional creation of software solutions is often termed *Software Engineering (SE)*, which can be defined as:

> the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.[1]

The term *Software Engineering* was first coined at the *NATO Science Conference* in 1968 (Naur & Randell, 1969), which marked the birth of the software engineering discipline.

[1] IEE (1990)

The conference, organized by the NATO Science Committee, was attended by many well-known computer science professionals of the time. Its aim was, in the committee's own words, to address, firstly, "the problems of achieving sufficient reliability in the data systems which are becoming increasingly integrated into the central activities of modern society", and, secondly, "the difficulties of meeting schedules and specifications on large software projects".[2]

Over the past four decades since the NATO conference, several diverse *software engineering methodologies* have been proposed and adopted. The history of these various methodologies will be described later in this chapter, in section 2.3. For the purposes of this dissertation, it is important that one such "family" of software engineering methodologies—the so-called *Agile* software methodologies—be defined. According to Ambler (2008), Agile software methodologies are characterised by their:

> iterative and incremental (evolutionary) approach to software development which is performed in a highly collaborative manner by self-organizing teams with 'just enough' ceremony that produces high quality software in a cost effective and timely manner which meets the changing needs of its stakeholders.

The most prominent of the Agile methodologies (TNM, 2000), *Extreme Programming (XP)*, is a lightweight methodology based on a set of core values, principles and practices for co-located teams of varying sizes who develop software in the face of vague or rapidly changing requirements. XP was first formulated by Kent Beck while he was the project leader for the *Chrysler Comprehensive Compensation (C3)* payroll project (CCC, 2008), which began in 1995. By 1997, XP had been formalised during the work on the C3 project and in 1999, the first book describing the methodology, *Extreme Programming Explained*, was published by Beck. The second edition of this book followed in 2005. It will be argued later in this dissertation that Beck prescribes a *revolutionary* approach towards adopting XP in the first edition of his book, in which **all** XP values, principles and practices should be adopted in order to gain the full benefit of XP, whereas in the second edition of his book, he suggests an *incremental* approach towards adopting the XP methodology's values, principles and practices. The former approach, it will be argued, is more similar to the Kuhnian method of *scientific revolutions*, whereas the latter approach is more similar to a Popperian approach

[2]Coincidentally, the 40^{th} anniversary of the NATO conference is being commemorated this year.

of *piecemeal social engineering*. This is despite the fact that the second edition, like the first, contains an explicit citation of Kuhn's *The Structure of Scientific Revolutions* in the annotated bibliography.

Evolution and *revolution* are two further key terms that will be used throughout this dissertation since they relate respectively to Popper and Kuhn's contrasting philosophies. In the most general sense, *evolution* can be defined as "a gradual development, especially to a more complex form"[3] and *revolution* as "a far-reaching and drastic change, especially in ideas, methods, etc."[4]. Evolutionary change involves small cumulative steps over a long duration, whereas revolutionary change occurs in sudden and radical "leaps". Neither necessarily implies progress.

Finally, two terms related to evolution and revolution are those of *emergent evolution*, a philosophical "doctrine that, in the course of evolution, some entirely new properties, such as life and consciousness, appear at certain critical points, usually because of an unpredictable rearrangement of the already existing properties"[5] and *punctuated equilibrium*, namely the "theory that evolution proceeds mainly in fits and starts, rather than at a constant rate (gradualism)"[6]. These latter two terms will refine our understanding of the opposition between the terms *evolution* and *revolution*.

Having introduced and defined the key terms for this dissertation, the following two sections will provide respective accounts of the history of the methodologies of science and software engineering, in order to contextualise the philosophical discussions in the remainder of this dissertation.

2.2 History of Scientific Method

This section will trace the origins of science and scientific method from the ancient Greek philosophers to the present day in order to provide a broad and comprehensive philosophical background for the discussions in the remainder of this dissertation. There are several reasons for starting as far back in history as with the ancient Greek philosophers, namely:

1. It will help to contextualise Kuhn and Popper's philosophy more broadly, since not

[3]Makins *et al.* (1991) (p. 539)
[4]Makins *et al.* (1991) (p. 1325)
[5]Makins *et al.* (1991) (p. 509)
[6]Bullock & Trombley (1999) (p. 711)

all readers of this dissertation may be familiar with the philosophical background of science.

2. Science and philosophy were originally inextricable, and modern scientific method must be understood as a departure from what came before.

3. Modern physics and cosmology are considered by many to be paradigm cases of knowledge to which all other disciplines should aspire, however imperfectly.

4. As will be discussed in chapter 3, "Related work", several other software engineers make use of the ideas of Plato and Aristotle.

5. Plato, Socrates and the pre-Socratics are important to Popper in terms of epistemology, cosmology and ethics, and all of these are relevant to this dissertation.[7]

6. The philosophy of Plato is important to Pekka Himanen and Eric Raymond, two leading figures of the open source software community, whose work will be outlined in Appendix F.

The history of scientific method is dramatically illustrated by the history of cosmology, since this was the earliest branch of science to mature in the West and its development was central to the changing world views of different historical periods. The authority of Plato and Aristotle and of the medieval Church caused both science and philosophy to stagnate in the Middle Ages and it was only with the advent of the Renaissance that thinkers began to escape medieval scholastic philosophy and that science started to become a separate discipline from philosophy.[8] The history of science is thus in part a history of the battle between critical thought and authority.[9] The irony seems to be that in achieving epistemological hegemony in the modern world, science has become endowed with the very type of unquestionable authority of religion from which it originally struggled to liberate itself. Thus an understanding of the nature and history of the methodology and ideology of science is crucial for anyone concerned about the place of science in an open society.

[7] Popper (1963) (pp. 183-185)
[8] Russell (1961) (p. 483)
[9] This is, of course, the liberal and Enlightenment perspective of progress in science, which has increasingly come under attack in the 20^{th} century. Feyerabend, for instance, revises the liberal view of the clash between Galileo and the Church. (Feyerabend, 1988) (ch. 11)

The tension between the authoritarian and libertarian elements in science are most clearly embodied in the opposing philosophies of Kuhn and Popper.[10]

Science and philosophy were initially inextricable and originated in the critical tradition of the pre-Socratic Greek philosopher-scientists. The method of these early thinkers was deductive and rationalist rather than inductive and empiricist: they tried to deduce all observable phenomena from general principles[11]. The strength of the ancient Greek thinkers of the critical tradition was their willingness to formulate new, bold hypotheses of the universe without deferring to authority;[12] the weakness was their failure to engage in painstaking empirical observation, thus revealing a bias towards deductive and against inductive logic. The deductive and geometric method was best exemplified in Euclid's *Elements*.[13]

Like the pre-Socratic philosophers, Plato tried to deduce an account of the physical world from a set of general principles in his late dialogue the *Timaeus*.[14] His ideas fed into the Aristotelian-Ptolemaic, geocentric theory of the universe that was to predominate for about 1500 years, hampering scientific progress in cosmology. Likewise, Aristotle's deductive system of syllogistic logic dominated European thought for about 2000 years, thereby inhibiting the growth of that discipline[15]. Furthermore, the Catholic Church's adoption of the Aristotelian philosophy of Thomas Aquinas as its official philosophy in the 13th century[16] further hampered the development of science, since any novel scientific theory was considered a challenge to the authority of the Church and was thus actively discouraged.

Renaissance philosophy was a revolt against the arid scholasticism of medieval philosophy, thanks partly to the rediscovery of Plato's works other than the *Timaeus*.[17] Although the "natural philosophers" of the period were considered both scientists and philosophers, the Renaissance saw the beginnings of the separation of science from philosophy. The move from medieval to modern was most dramatically represented by the Copernican revolution in cosmology, culminating 150 years later in the Newtonian paradigm, replacing the Aristotelian-Ptolemaic system completely. These scientists combined theoretical boldness

[10]Fuller (2003) (p. 11)
[11]Russell (1961) (pp. 55, 221)
[12]Russell (1961) (pp. 25, 89)
[13]Russell (1961) (p. 221)
[14]Russell (1961) (p. 157)
[15]Russell (1961) (p. 212)
[16]Russell (1961) (pp. 444-445)
[17]Russell (1961) (p. 483)

with careful observation. Their method, the so-called inductivist method of modern science, was first formulated by Francis Bacon.[18]

Although his ideas were faulty, Bacon's inductivism is still widely considered to be the method of science. It involves the accumulation of facts through careful observation, until a pattern can be discerned and a general statement derived from the empirical evidence. After sufficient evidence has been accumulated, the general statement can be said to be proved and its status as a law of nature established. Thus, in contrast to the ancient Greeks and the continental rationalist philosophers who deduced the observable from general principles, the inductivist method proceeds in the opposite direction, inducing general laws from observable facts.

David Hume was perhaps the most outstanding critical thinker in the history of philosophy. He applied devastating critiques to the concepts of induction, causation, the self and miracles, amongst others. Concerning induction, he argued that no number of particular observations can justify (prove) an empirical generalisation, meaning that no supposed law of nature, even the most successful scientific theory, can ever be certainly true. He showed that an inductive inference is never valid. Bertrand Russell believed that Hume's critique of induction (and causation) showed not merely the limitations of empiricism and the physical sciences, but demonstrated the bankruptcy of eighteenth-century reasonableness and the self-refutation of reason.[19]

Although Hume's critiques were initially ignored, they eventually came to the notice of Emmanuel Kant, who was spurred on to achieve a synthesis of the rationalist and empiricist traditions. Empiricists had rejected the rationalist belief in self-evident, innate ideas, arguing that there were only two kinds of true statement, the analytic statements of mathematics and logic, which are (deductively and necessarily) true by definition, and the synthetic statements of the empirical sciences, which can be verified by experience and are only contingently (accidentally) true. They considered analytic truth to be *a priori* (namely, to precede experience) and synthetic to be *a posteriori* (to follow experience). Kant proposed a third alternative, that of the synthetic *a priori*, namely the most fundamental categories of thought that any rational mind must possess in order to experience anything at all.[20] For instance, he believed that Euclidean geometry provided the necessary

[18] Russell (1961) (pp. 526-530)
[19] Russell (1961) (pp. 645-646)
[20] Russell (1961) (pp. 679, 685)

basis for our category of space. He considered his discovery of the synthetic *a priori* to be the Copernican revolution of philosophy (or epistemology).[21] The discovery of alternative systems of geometry in the 20^{th} century, however, have apparently confirmed empiricists' doubts about Kant's epistemology.

Despite the spectacular success of the Newtonian paradigm, the theory was eventually superceded by Einstein's relativity theory, not, of course, without initial resistance. This revolutionary displacement of a well-established and well-confirmed theory with a new one had massive implications for a theory of scientific method. However, rather than creating doubt in people's minds as to science being the final authority on epistemological matters, many people considered the new paradigm to have the final word.[22]

The high point of scientism occurred in Vienna in the 1920s and 1930s. The Vienna Circle consisted not only of philosophers but also of leading physicists and its philosophy was known as logical positivism. It was an attempt to combine the logical rigour of modern mathematics and logic with the empiricism of Hume, Comte and Mach.[23] Logical positivists elevated science, most notably physics, above all other disciplines and endowed it with an epistemological authority previously possessed by the medieval Church. They argued that science was value-neutral and involved a purely rational progression towards truth by means of the inductive confirmation of general laws of nature through the steady accumulation of empirical evidence.

Even though the school dissolved in the late 1930s, it had a profound influence on subsequent Anglo-American philosophy, as was most clearly stated in A.J. Ayer's *Language, Truth and Logic* (Ayer, 1971), first published in 1936. Logical positivism adhered to the analytic-synthetic distinction and posited verificationism as the criterion of meaningful statements. Sense experience was therefore the foundation of knowledge and its ultimate authority. Thus for the positivists the statements of theology, metaphysics, aesthetics and ethics were considered meaningless and only the statements of science were meaningful. However, they were nonetheless saddled with the problem of induction, that is the justification of empirical generalisations, and with the embarrassing fact that everything they wrote about science, according to their own criterion of verifiability, was meaningless since their statements could not themselves be empirically verified. Furthermore, their fixation

[21] Russell (1961) (p. 680), Popper (1963) (p. 244)
[22] Popper (1963) (p. 258)
[23] Ashby (1964) (pp. 493-494)

on logic and language hearkened back to the sterility of medieval scholastic philosophy. With the rise of Nazism in Europe, many of the logical positivists escaped to the United States, where they had a profound influence on the American academic establishment.

Karl Popper has often been incorrectly identified as a logical positivist.[24] In fact, although he was associated with the Vienna Circle, he was one of its most powerful critics and claimed to have killed logical positivism. He pointed out that observation can never be neutral but must always be theory-laden, otherwise scientists would not be able to select the facts relevant to their research. Indeed, he argued that a scientist always starts with a problem to be solved and then formulates several hypotheses as tentative solutions to the problem. Then, rather than attempting to confirm the hypotheses, the scientist tries to eliminate the false hypotheses through rigorous testing and error elimination. The hypotheses that survive the severest testing are not then considered verified, but remain tentative approximations to the truth. Thus Popper rejected the foundationalist's quest for indubitable foundations of knowledge which placed him in the tradition of fallibilists like Kant and Peirce[25].

Popper also rejected the positivists' verifiability as a criterion of meaningfulness, and instead propounded falsifiability as a criterion for the demarcation of science from pseudo-science.[26] He argued that what differentiates scientific from pseudo-scientific generalisations (or theories) is the fact that the former can in principle be falsified (or refuted) whereas the latter are closed to refutation. Popper agreed with Hume that induction is never valid, that no general empirical statement (law of nature) can be verified. In fact, he went further and claimed that inductive logic simply does not exist but that science uses the hypothetico-deductive method instead.[27] He made much of the facts of deductive logic that while no number of confirmations is sufficient to prove an empirical generalisation, only one counter-instance is sufficient to disprove a generalisation. Popper was opposed to all forms of authority and foundationalism, even in science, arguing that no theory can be established beyond doubt. He argued that the scientific attitude is essentially critical, as exemplified by the critical tradition of the pre-Socratic Greek philosopher-scientists. Indeed, he called his ideas "critical rationalism" and maintained that the free pursuit of knowledge is best

[24] Quinton (1964) (p. 550)
[25] Miller (1983) (p. 10)
[26] Popper (1959) (p. 18)
[27] Popper (1963) (p. 73)

nurtured in a democracy or "open society".

Thomas Kuhn, the anointed American philosopher of science of the Cold War[28], started out as a physicist but later turned to the history of science, writing a substantial work on the Copernican revolution. He discovered that the actual history of science does not bear out the characterisation of science according to the positivists and Popper. He showed that the sciences should be seen as complex wholes or structures and that the focus on verifying or falsifying hypotheses is too narrow. His key concepts were those of *scientific revolutions* and *paradigm shift*, which he developed in his highly influential work, *The Structure of Scientific Revolutions* (Kuhn, 1962). By paradigm he meant both an exemplary piece of research and the *disciplinary matrix* or epistemological framework within which all scientists in a common field practise and which is usually tacitly assumed and uncriticisable. He argued that there can be no neutral collection of evidence but that all research is guided by the paradigm in which scientists work. Furthermore, he argued that there is no cumulative, piecemeal progress in science as imagined by the positivists and Popper, but merely the revolutionary replacement of an old paradigm with a new one, the different paradigms being *incommensurable*.

Kuhn used the term *puzzle-solving* rather than Popper's *problem-solving* to describe the activity of scientists working within a paradigm, the paradigm itself not being subject to criticism. "Most actual science—what Kuhn calls 'normal science'—consists of little more than the technical work of fleshing out the paradigm's blueprint."[29] Contra Popper, he pointed out that scientists do not seek rigorously to refute their hypotheses but rather tend to ignore anomalies until so many of them accumulate that they can no longer be ignored and the practitioners within the paradigm are thrown into a crisis. It is only during this period of crisis that scientists behave in the way Popper described, creatively formulating bold hypotheses and rigorously testing them, in order to find a new paradigm in which to operate. Also in contrast to Popper, Kuhn argued that truth is not objective, but merely the consensus of the scientists within a specialised scientific community. For Kuhn, scientific communities were an unquestionable, authoritative élite. According to Fuller (2003), Kuhn also encouraged a type of contingent foundationalism in relation to paradigms. For Popperians, Kuhn had retained "the most objectionably conservative features of logical

[28]Fuller (2003) (p. 32)
[29]Fuller (2003) (p. 19)

positivism", namely, the assumption that "science requires stable foundations for both legitimising and directing inquiry" (Fuller, 2003).

Imre Lakatos attempted to achieve a synthesis between the ideas of Popper and Kuhn in his idea of scientific research programmes. Like Kuhn, he recognised the fact that the sciences had to be considered as complex, organised structures.

> A Lakatosian research programme is a structure that provides guidance for future research in both a positive and a negative way. The *negative heuristic* of a programme involves the stipulation that the basic assumptions underlying the programme, its *hard core*, must not be rejected or modified. It is protected from falsification by a *protective belt* of auxiliary hypotheses, initial conditions, etc. The *positive heuristic* is comprised of rough guidelines indicating how the research programme might be developed. Such developments will involve supplementing the hard core with additional assumptions in an attempt to account for previously known phenomena and to predict novel phenomena. Research programmes will be *progressive* or *degenerating* depending on whether they persistently fail to lead to the discovery of novel phenomena.[30]

The hard core of a research programme helps to maintain its internal coherence despite its developing complexity. When research programmes are in competition with each other, the more progressive one will be chosen.

Paul Feyerabend, like Lakatos, was a former student of Popper's but, unlike Lakatos, became one of Popper's keenest critics. His statement concerning scientific method that "...there is only *one* principle that can be defended under *all* circumstances and in *all* stages of human development. It is the principle: *anything goes*"[31] should not be taken too literally since he was not really an epistemological anarchist, but merely claimed that no single scientific method should be prescribed, since that may constrain future research and since the history of science is too complex to be reduced to any single method. Thus he approved of the fact that Lakatos did not prescribe any method in his scientific research programmes. But he also agreed with Kuhn that competing paradigms are incommensurable and, therefore, that one cannot speak of progress in science. Like Kuhn, he emphasised the importance of subjecting general accounts of science to the facts of history. Finally, unlike

[30] Chalmers (1982) (p. 80)
[31] Paul Feyerabend *Against Method* (London: Verso, 1988), p. 19.

all the preceding philosophers of science, Feyerabend questioned the assumption that science is the paradigm of knowledge and rationality.

This section located the origin of science and scientific method in ancient Greek philosophy. It then traced the liberation of science from philosophy and religion in the Renaissance period, culminating in the final triumph of science in the Newtonian cosmology in the 18^{th} century Enlightenment, thus explaining the present epistemological hegemony of science and empiricism in the modern world. However, in sketching the increasing critical awareness of scientific method from Hume's devastating critique of induction up until the present day, this section questioned the claim that science represents the paradigm case of knowledge and rationality. Finally, it also exposed science as an ideology, betraying the critical role that Popper believed it should have in an open society.

2.3 History of Software Engineering Culminating in Agile Methodologies

The aim of this section is to provide a historical account of the major events in the software engineering discipline over the past four decades, in order to contextualise contemporary software development and to inform the philosophical discussions in the remainder of this dissertation. In particular, the historical shifts from one type of software methodology to another will be discussed and an attempt will be made to see how far these shifts in methodology can be regarded as *revolutionary* or *evolutionary*, since these are two important concepts to the respective philosophies of Kuhn and Popper.

Any software engineering endeavour involves three fundamental parts: it follows a *methodology* or "software process", it involves the use of software *tools*, and it produces software *artefacts*. Mutually interdependent changes in all three of these—processes, tools and artefacts—have occurred over the past few decades.[32] But rather than comprehensively surveying the major changes that have occurred in all three of these, the focus in this section (as well as throughout this dissertation) will be on the major changes that have occurred in software *methodology*. The reason for narrowing the focus to software methodology is motivated by the central aim of this dissertation, which is to provide concrete analogies

[32]For example, the introduction of compilers (a software artefact) impacted radically on the software process, producing, in turn, more complex software artefacts.

between particular types of software methodologies and particular scientific methodologies.

Pre 1960s

In the early years of computing, there was no clear notion of a software process or methodology. Computer programs were usually embedded as part of the mechanical hardware and were simply developed in an *ad hoc* or *unstructured* fashion based on the available technology. Initially this meant hand-wiring connections, then coding by hand in machine language, and later in mnemonic code. Some early programming languages like FORTRAN, LISP and COBOL were created in the 1950s and provided the impetus for high-level programming languages, and the consequent shift away from machine dependent languages. This shift was also facilitated by early tools such as assemblers and interpreters as well as optimising compilers. During this period, the principal stimulus for adopting a methodological standpoint about how software should be developed was the need to control the ever-growing *complexity* associated with software development, notwithstanding the constantly improving tools and notations for doing so. The need to improve quality and productivity were additional driving forces for adopting a rigorous methodology.

1960s

During the 1960s, software engineering began to be established as a distinct academic discipline and increasingly complex programs containing millions of lines of code were being developed to support critical systems, usually for military projects. The size of project teams also grew dramatically. The U.S. Department of Defense (DoD), pressurised by the threat of nuclear attack since the start of the Cold War in 1945, created the Advanced Research Projects Agency Network (ARPANET), which is now widely considered to be the predecessor to the modern Internet. ARPANET was based on the revolutionary idea of a packet switching network in which each packet of data could be routed to the destination independently of other packets, thereby minimising loss of data in transit if the network was under attack. It also facilitated the decentralisation of information, which, in turn, helped to reduce the extent of data loss in case a specific site was attacked. Due to the large influence that the DoD had on the software engineering discipline as a result of the ARPANET, many software engineers worked on projects funded by the military during the 1960s.

Another landmark in the 1960s was the publication of Dijkstra's influential paper, *"Go To Statement Considered Harmful"* (Dijkstra, 1968), in 1968. In his paper, Dijkstra identified a major source of complexity in early software development and argued that unrestricted GOTO statements should be discarded from higher-level languages because they complicate the task of analyzing and verifying the correctness of programs. Initially, Dijkstra's ideas were resisted and considered radical. However, the subsequent rapid and universal adoption of his *structured programming* as a development style had all the hallmarks of revolutionary change.

Also in 1968, the term *software engineering* was coined by F.L. Bauer during the influential *NATO Software Engineering Conference*, which, as described at the start of this chapter, marked the start of the profession of software engineering. The main theme of the conference was the *software crisis* which started in the mid 1960s when, for the first time in computer history, the cost of software started to exceed the cost of hardware. The frequency of over-budget and over-schedule projects, some of which also caused damage to property and even resulted in the loss of life, increased dramatically. While some believe that the software crisis ended in the mid 1980s (HSE, 2008), since it is unrealistic for a discipline to remain in crisis for more than twenty years, others believe that the crisis still remains at the heart of the discipline today (SEA, 1998). Reflecting on the use of the term *software engineering* almost thirty years after the NATO conference in a seminar on the history of software engineering, Mary Shaw, a leading researcher in the field, said that, to a large extent, the term "software engineering", was still "a phrase of aspiration, not of description" (Shaw, 1996).

1970s

As projects grew in size, it was perhaps natural for project managers with engineering backgrounds to look to their traditional style of management in an attempt to control the challenge of ever increasing system complexity.[33] The influence of the classical engineering discipline on the software engineering discipline meant that the life cycle of a software project was viewed as a sequence of discrete phases, each of which needed to be articulated, documented, executed and signed off **before** proceeding to the next phase. More specifically,

[33]The website http://www.sereferences.com/software-failure-list.php lists a number of spectacular software failures, which are all due to the intrinsic complexity of modern software systems and the limited ability of the human mind to cope with such high levels of structural and behavioural complexity.

once the requirements had been elicited, documented and signed by the client, an analysis phase was entered. Here the task was to partition the implementation of a software system, *a priori*, into several modules that could be written by different software developers.[34] After the analysis phase, further phases followed including design, implementation, testing, and so on. This phased approach towards software development was eventually termed the *Waterfall Model* by Royce in his well-known paper *Managing the Development of Large Software Systems* (Royce, 1970) and it hallmarked a significant change from informally to formally managed projects. Such a change clearly involved an important conceptual shift, and in this sense, the adoption of the *Waterfall Model* can be considered *revolutionary*. However, since there were relatively few large projects in the emerging software industry at the time, the uptake in applying the *Waterfall Model* was gradual. Nevertheless, the model ultimately became the orthodoxy of the day, and was considered mandatory for proper management of large software projects, especially "by such large software development houses as those employed by the DoD and NASA" (WMO, 2008). The authoritarian roots of this methodology undoubtedly derive from the influence these military and governmental organisations had on the *Waterfall Model*, a point which will be elaborated on later in section 6.1.2 of this dissertation. The *Waterfall model*, which is depicted in Figure 2.1 below, will also be used as a contrasting reference model when discussing Agile methodologies later in this dissertation. Essentially, it can be criticised for applying industrial techniques to the software process, thereby reducing software engineers to interchangeable units or expendable resources. Another criticism of the *Waterfall model*, is its assumption that software can be comprehensively designed a priori, and that such designs will not require much change throughout the project life cycle.

1980s

The 1980s saw the rise of personal computers and the consequent rise of proprietary consumer software. In 1983, in response to the widespread commercialisation of software, which saw a dramatic increase in proprietary software under restrictive licensing terms, Richard Stallman started the *Free Software Movement* which culminated in the creation of a non-profit organisation called the *Free Software Foundation (FSF)*. The aim of the FSF was to

[34] This task was supported technically by advances in programming languages, where modularisation was becoming an ever-stronger theme. For example, *Modula* (Wirth's nomenclature for his programming language that followed on Pascal), bears testimony to this. Although never implemented, its successor, *Modula-2*, was widely used in the early 1980's, and *Modula-3* was also available in the 1990s.

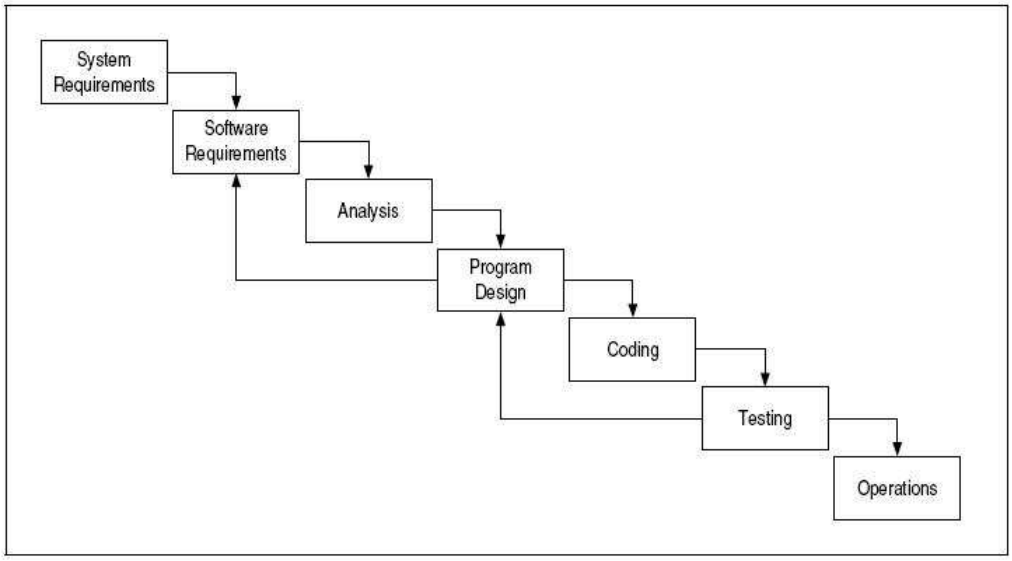


Figure 2.1: Waterfall Model (Royce, 1970)

promote the universal freedom to distribute and modify software without restriction. Towards the end of that decade, Stallman's approach was merged with the *Free/Libre/Open Source Software (FLOSS)* movement which granted users the right to study, change, and improve software source code. This movement can be considered revolutionary (DiBona *et al.*, 1999), at least in terms of a shift in mindset towards a radical alternative to proprietary software.

By the late 1980s, the inappropriate nature of the *Waterfall Model's* rigid, phased software process had become increasingly apparent, not least because of the large number of failed software projects. Particularly troublesome was the dogmatic requirement of committing to one phase before proceeding to the next. As a result, there was a strong shift towards *Iterative Incremental Development (IID)*, as advocated by Boehm. He proposed the *Spiral Model* (Boehm, 1988), which emphasised the notion of developing software in *cycles* or *iterations*, as Figure 2.2 depicts. Each cycle of the spiral, according to Boehm, begins with an identification of the objectives of a part of the product being elaborated, the alternative means of implementing this part of the product and the constraints imposed on the application. The next step is to evaluate the alternatives relative to the objectives and constraints, thereby identifying risks. Prototyping, simulation and benchmarking are used as risk resolution techniques. The following step is determined by the relative remaining risks. Finally, each cycle is completed by a review involving the primary stakeholders of the

product. This review covers all products developed during the previous cycle, including the plans for the next cycle, and the resources required to execute them[35].

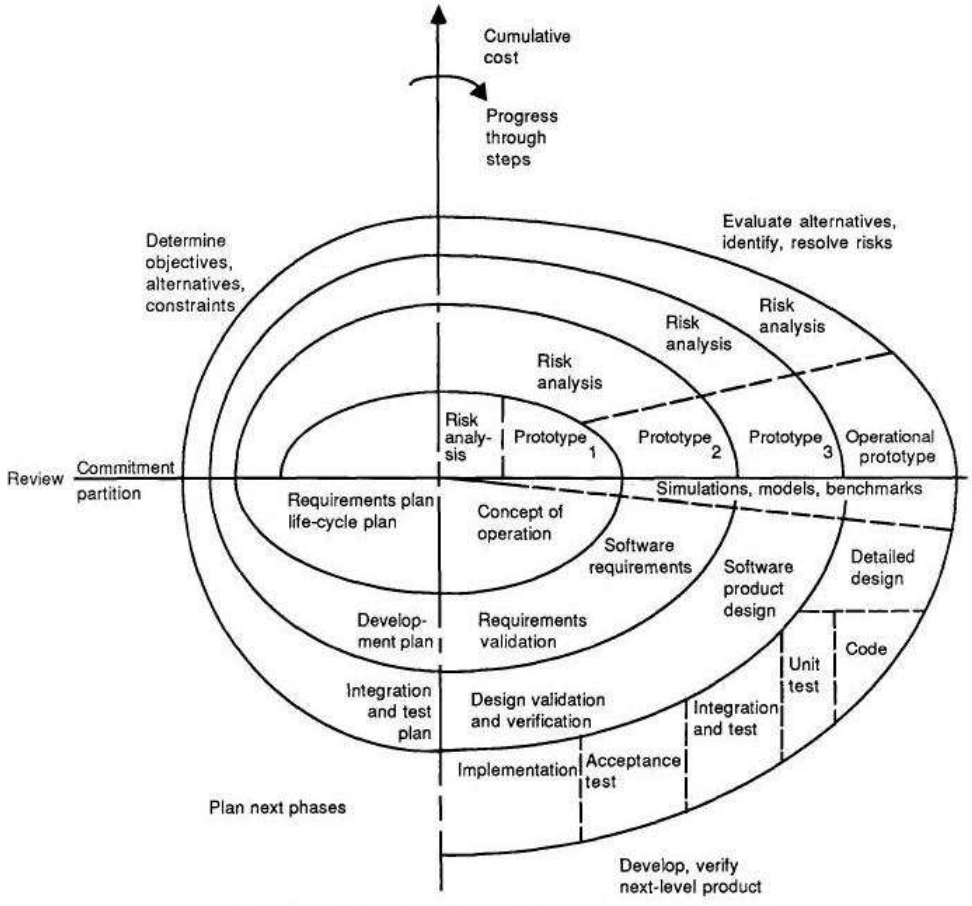


Figure 2.2: Spiral Model (Boehm, 1988)

This shift away from Waterfall-like methodologies relied largely on *object-oriented programming (OOP)* which was, once more, supported by appropriate new programming languages. In essence, the modularisation of software changed. A module no longer represented a set of similar tasks that changed the state of a large a number of heterogeneous objects in a domain. Instead, the focus shifted to identifying homogeneous objects from the same software class, whose state could be encapsulated and manipulated by functions particular to that class of objects. Software objects began to reflect so-called "real-world" objects in the problem domain, which meant that the problem and solution spaces were brought closer together. This, in turn, gave rise to the concept of *design patterns*—repeatable solutions to commonly occurring problems in software development. The design patterns movement

[35]Adapted from Boehm (1988) (p. 65).

in software engineering was inspired by the architect, Christopher Alexander, who applied patterns of successful solutions to the design of towns and buildings. The shift that OOP ensured away from *a priori* commitment to all artefacts to be produced in a given phase of development, to gradual commitment to smaller parts of the project, coupled with a new way of partitioning the software into objects could arguably be classified as revolutionary, although the uptake of OOP was also somewhat gradual.

For almost two decades, solving the software crisis was of central importance to organisations producing software. Virtually every new technology and practice formulated between 1970 and 1990 was proclaimed to be the "silver bullet" for solving the software crisis. However in 1987, Brooks published an influential article called *No Silver Bullet: Essence and Accidents of Software Engineering* (Brooks, 1987) in which he distinguishes *essential* difficulties in software engineering from *accidental* difficulties. He argues: "But, as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity". Nowadays, most software engineers accept Brooks' findings and agree that the software engineering discipline appears to be too complex and diverse for a single "silver bullet" technology or practice to solve most major problems in the discipline.

1990s

The 1990s saw the price of hardware plummet and the capacity of computer hard drives and memory increase dramatically, resulting in even more complex software systems and exacerbating the software crisis further. The increased popularity of mobile devices containing embedded software applications also meant that software became more widely available.

The dawn of the *Internet era* at the start of the 1990s, no doubt, triggered by the ending of the Cold War in 1989 and the resultant unrestricted globalisation, saw followers of the traditional methodologies struggling to adapt to rapid shifts in the modern business world. In particular, they seemed unable to cope with the challenges that arose from the Internet era like shortened market time windows, rapid technological change, increasingly volatile business requirements and the rising complexity of software systems. Ultimately, traditional methodologies seemed unable to satisfy software development needs in the Internet era, which required methodologies to be flexible in the face of rapid change. The

worsening software crisis provided the impetus for various "lightweight" methodology proposals including *Adaptive Software Development (ASD)*[36], *Extreme Programming (XP)*[37], *Scrum*[38], *Crystal*[39], *Feature-Driven Development (FDD)*[40] and *Dynamic System Development Method (DSDM)*[41]. These methodologies remained based on IID but were less formal on the so-called "ceremonial" requirements and were fundamentally focused on accommodating *change*. The Internet era also stimulated *distributed* software development, which, in turn, rekindled an interest in FLOSS.

In 1991, the Software Engineering Institute (SEI) at Carnegie Mellon University released the *Capability Maturity Model (CMM)*, a *process model* used to assess, on a level of one to five[42], the maturity of an organisation's processes. The SEI was originally established by the DoD in 1984, and the CMM resulted from a study at the SEI funded initially by military research into why subcontractors of military projects ran over-budget, and often over-time as well. The hope was that the CMM would provide a yardstick for the objective evaluation of software subcontractors for the military. Despite being originally intended for military projects, the CMM was widely adopted by the broader software engineering community. Later, however, the CMM was largely superseded by the *Capability Maturity Model Integration (CMMI)*, which evolved from it, and was released in 2006. The CMMI has the same five maturity levels as the CMM, but its *key process areas (KPAs)* differ from those of the CMM, being more aligned with modern iterative development techniques and modern software management best practices. Both models seem to assume that organisations start on level 1, where the software process is *ad hoc*, and only later learn the disciplines of developing good software. Due to the widespread popularity of both the CMM and the CMMI model, especially as the outsourcing of software projects began to grow, their uptake could arguably be considered *revolutionary*, especially since they are not used exclusively in software projects, but have been applied more widely as models for assessing general process maturity. To complement the CMM, the Testing Maturity Model (TMM) (Burnstein *et al.*,

[36] ASD was developed by Highsmith and set the 'philosophical' benchmark for the other Agile methods. It evolved from Rapid Application Development (RAD).

[37] XP was developed by Beck and comprises a set of core values, principles and practices. XP is said to have established the popularity of the Agile methods.

[38] Scrum, a methodology grounded in complexity theory, was created by Schwaber using a rugby metaphor.

[39] The Crystal family of methodologies, created by Cockburn, focus on categorising projects by criticality.

[40] FDD was created by Coad and is a model-driven process consisting of five steps.

[41] DSDM is a framework developed in the 1990s by the DSDM Consortium and is based on RAD.

[42] The five maturity levels are *initial* (level 1), *repeatable* (level 2), *defined* (level 3), *managed* (level 4), and *optimising* (level 5). The aim for organisations is to move towards level 5, where each level represents an increased maturity, as a result of following sound engineering and management practices.

1996) was created by the Illinois Institute of Technology's Computer Science Department in 1996 as a model for test process improvement. The TMM, like the CMM, has five maturity levels. These levels define the testing capabilities of an organisation and indicate which levels an organisation should focus on in order to improve its testing process. Later, the TMM*i* (TMM, 2002), guided by the work on the CMMI, was created in 2002 using the TMM framework as a basis.

In 1993, the *Rational Unified Process (RUP)* was introduced by Kruchten[43]. RUP evolved from IID but also had roots in Boehm's *Spiral Model*. It was designed to have an underlying object-oriented model using the international standard *Unified Modeling Language (UML)*, a graphical notation used to create an abstract model of software systems. Modeling helps to improve software development by encouraging early reasoning about the design and architecture of a software system.

Towards the end of the decade, there was a strong drive towards advancing the professionalism in software engineering, towards formalising the field, and towards providing a standard for teaching and practising software engineering. To this end, in 1998, the Institute of Electrical and Electronic Engineers (IEEE) and the Association for Computer Machinery (ACM) together established the *Software Engineering Code of Ethics and Professional Practice (SECEPP)*[44], the aim of which was to detail the ethical and professional obligations of software engineers, with "Public Interest" being at the centre of the code. Another step towards promoting professionalism in software engineering was the introduction of the *Software Engineering Body of Knowledge (SWEBOK)*, once again endorsed by the IEEE and the ACM. Perhaps one of the main motivations for attempting to improve professionalism through the SECEPP and the SWEBOK, was the belief that unskilled and irresponsible software engineers may harm the software development industry.

2000s

The outsourcing of software projects became widespread in the 2000s, which changed the nature and focus of software engineering projects. However, arguably the most significant event to take place in software engineering so far this decade occurred in February 2001 when seventeen representatives of the "lightweight" methodologies met in Salt Lake City,

[43]Kruchten (2003)
[44]Gotterbarn (1999)

Utah, for the "Lightweight Methods Summit" to discuss the commonalities of their methodologies. This summit was initiated by an earlier meeting in 2000 called by Beck, the aim of which was to discuss the creation of an organization to drive and support the adoption of XP. The participants in the latter meeting suggested to broaden the scope of the proposed organization to include various "lightweight methods" and, as a result, the "Lightweight Methods Summit" was organised. During the summit meeting, the group of representatives agreed to adopt the term "Agile" instead of "lightweight" to represent their methodologies and they issued the *Manifesto for Agile Software Development* (Beck *et al.*, 2001a) containing a statement of the four central "values" underlying the Agile approach towards software development. The Manifesto emphasises:

> *Individuals and interactions* over processes and tools,
> *Working software* over comprehensive documentation,
> *Customer collaboration* over contract negotiation,
> *Responding to change* over following a plan.[45]

Later, in order to support these four values[46], a further dozen principles were formulated emphasising: prompt customer satisfaction, flexible response to change, frequent delivery of software, daily interaction between customers and developers, motivated individuals, face-to-face communication, measuring progress through working software, sustainable development, technical excellence, simplicity, self-organising teams, and reflection. The summit also marked the birth of the *Agile Alliance*, a non-profit organisation, which would go on to promote the Agile principles and values. Furthermore, it marked the beginnings of the so-called "Agile community", a loyal group of supporters driven by the values and principles of the Manifesto. This community continues to grow today with over four thousand members worldwide having joined the Agile Alliance since its inception in 2001 (AAL, 2008). An international conference is held annually in August, where the members of the Agile community get together to exchange information about Agile software development[47]. Another organisation that promotes Agile software development and project management is the international IT advisory firm, the Cutter Consortium, which was established in 1986.

[45] See Appendix A for a complete listing of the Agile Manifesto.
[46] Strictly speaking, these "values" would be more appropriately termed *maxims* since they do not relate to ethics as such but are, nonetheless, widely accepted without requiring further justification.
[47] The "Agile 2008 Conference" will be held in August in Toronto (A2C, 2008).

The members of this consortium also hold an annual "Summit" conference[48].

The Agile Manifesto spawned a movement in the software engineering discipline known commonly as the *Agile software development movement*. The adoption of Agile methodologies increased over the years and currently, according to Ambler (2007), indications are that Agile methodologies have crossed Moore's technology-adoption chasm, and gained mainstream acceptance. The popularity of Agile methodologies was contributed to significantly by Kent Beck's methodology called *Extreme Programming (XP)*. As described at the start of this chapter, XP was first formulated in 1995 by Beck while he was the project leader for the C3 project. Despite the project's eventual failure in 2000, it is still credited with the birth of the XP methodology. The three core components of XP—values, principles and practices—aim to enable projects to thrive on *change* throughout the project life cycle whether that change manifests itself in the form of vague or rapidly changing requirements, designs, or technologies. Indeed, the second edition of *Extreme Programming Explained* (Beck & Andres, 2005) which is subtitled *Embrace Change*, explains:

> This is the paradigm of XP. Stay aware. Adapt. Change.
> Everything in software changes. The requirements change. The design changes. The business changes. The technology changes. The team changes. The team members change. The problem isn't change, because change is going to happen; the problem, rather, is our inability to cope with change.[49]

Furthermore, Beck writes:

> Extreme Programming (XP) is about social change... [it] is my attempt to reconcile humanity and productivity...[50]

These two aspects, *humanity* and *productivity*, will be related respectively to the two main parts of this dissertation, the *ethical* and *epistemological*, and will, therefore, be central to the philosophical discussions throughout this dissertation. Regarding productivity, Beck points out that "Technique is also important. We are technical people in a technical field". With regard to humanity, he states "Good, safe social interaction is as necessary to successful XP development as good technical skills". The XP development model takes both of these aspects into account throughout the life cycle of a project.

[48]The *Summit 2008* conference will be held in May in Massachusetts (S20, 2008).
[49]Beck & Andres (2005) (p. 11)
[50]Beck & Andres (2005) (pp. 1-3)

From this description of Agile methodologies and, in particular, XP, it can be said that, in one sense, these methodologies are an *evolutionary* offshoot of iterative incremental processes. However, in another sense, they represent a radical, *revolutionary*, departure from all previous processes, in that they spurn an ideological commitment to being adaptable to change, and place strong emphasis on the efficient production of the final product while always taking cognisance of the human needs of the whole Agile team. Using Kuhn's notion of "scientific revolutions", chapter 4 will investigate whether Agile methodologies can be considered a revolutionary departure from the previous, traditional, methodologies.

At the present point in time, Agile software methodologies have certainly caught the attention of the software engineering community and have been adopted by many organisations. However, it remains to be seen whether they will become the new orthodoxy of tomorrow.

Subsequent to the popularisation of Agile methodologies, various *Unified Process* derivatives evolved, mostly from RUP, including the *Agile Unified Process (AUP)*, the *Open Unified Process (OpenUP)*, the *Enterprise Unified Process*, and so on. There has also been a trend, initiated by the prominent Agile advocate, Cockburn, towards "a methodology per project" approach (PPP, 1999), which discourages practitioners from committing a priori to a single methodology for all projects. Instead, Cockburn's approach encourages software engineers to make a rational choice between available methodologies depending on the specific requirements of each individual project.

To summarise this section on the history of software methodologies, it can be said that there have been considerably changes over the past four decades in software engineering processes (methodologies). Whether these changes can be classified as evolutionary or revolutionary, though, seems to be somewhat subjective. Perhaps the term *punctuated equilibrium*—introduced at the start of this chapter—describes more adequately their overall evolution, which has not proceeded at a constant rate but rather in fits and starts. Figure 2.3 provides a graphical summary of the changes in software methodology that are considered the most important for the discussions in this dissertation.

Finally, Figure 2.4 depicts a combined timeline of the most important historical events for this dissertation from those described in the history of scientific and software methodology sections in this chapter.

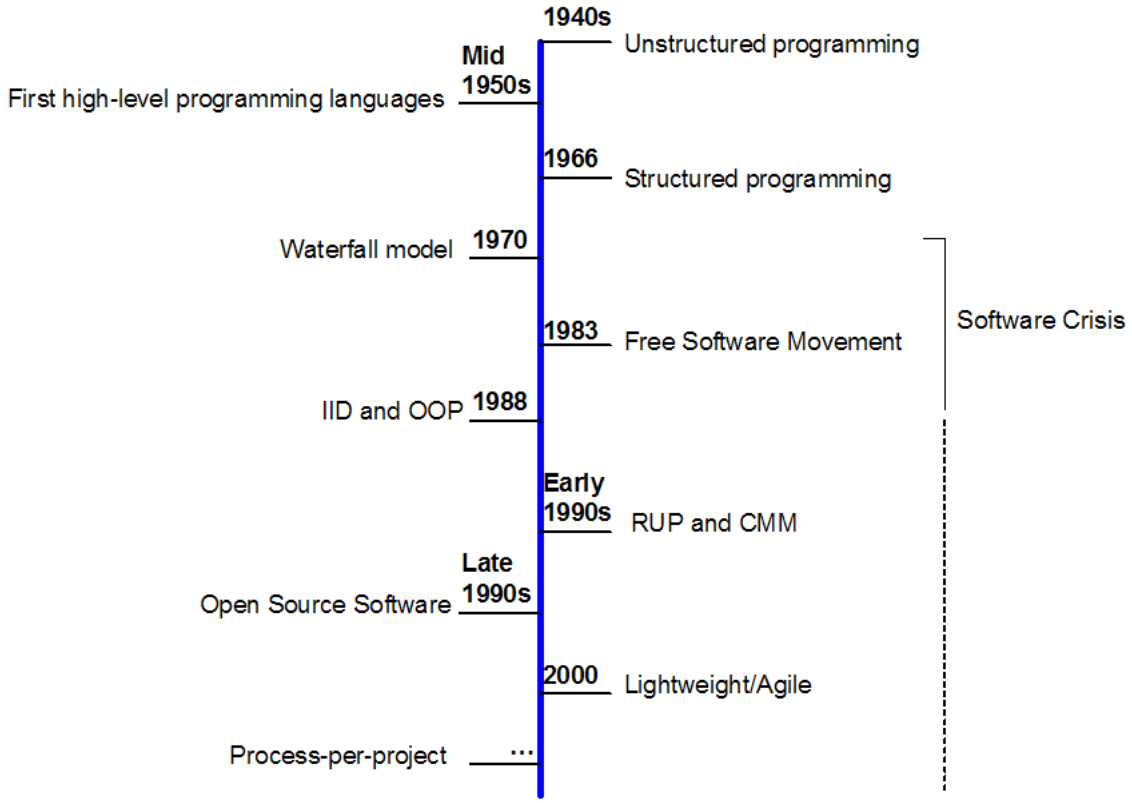


Figure 2.3: Timeline of Historical Changes in Software Methodology

2.4 Life Cycle of a Typical Agile Software Project

In this section, the activities that typically occur during a single iteration[51] of an Agile software project will be described. The aim of this section is to provide a concrete example for the philosophical discussions in the remainder of this dissertation. In particular, the iteration described here will be that of one within a typical *XP* project, since XP is the chosen representative of Agile methodologies in this dissertation, and will be used when analogies are drawn between the practices of a software methodology and those of a scientific methodology[52].

A typical XP project begins with a one to two week long "Planning Game" during the *exploration phase* of the project. The whole XP team, including the customer, is involved

[51] An iteration can be understood as a miniature software project within the overall software project. Each iteration includes all the tasks necessary to release a subset of the overall functionality in a short period of time. The motivation for implementing and releasing software in iterations is the deployment of a working software solution early, which will evolve over time. Some of the benefits of *incremental design* include increased flexibility and improved error elimination, as well as decreased cost and risk.

[52] All words in italics in this section represent either an XP value, principle or practice from the second edition of *Extreme Programming Explained* (Beck & Andres, 2005).

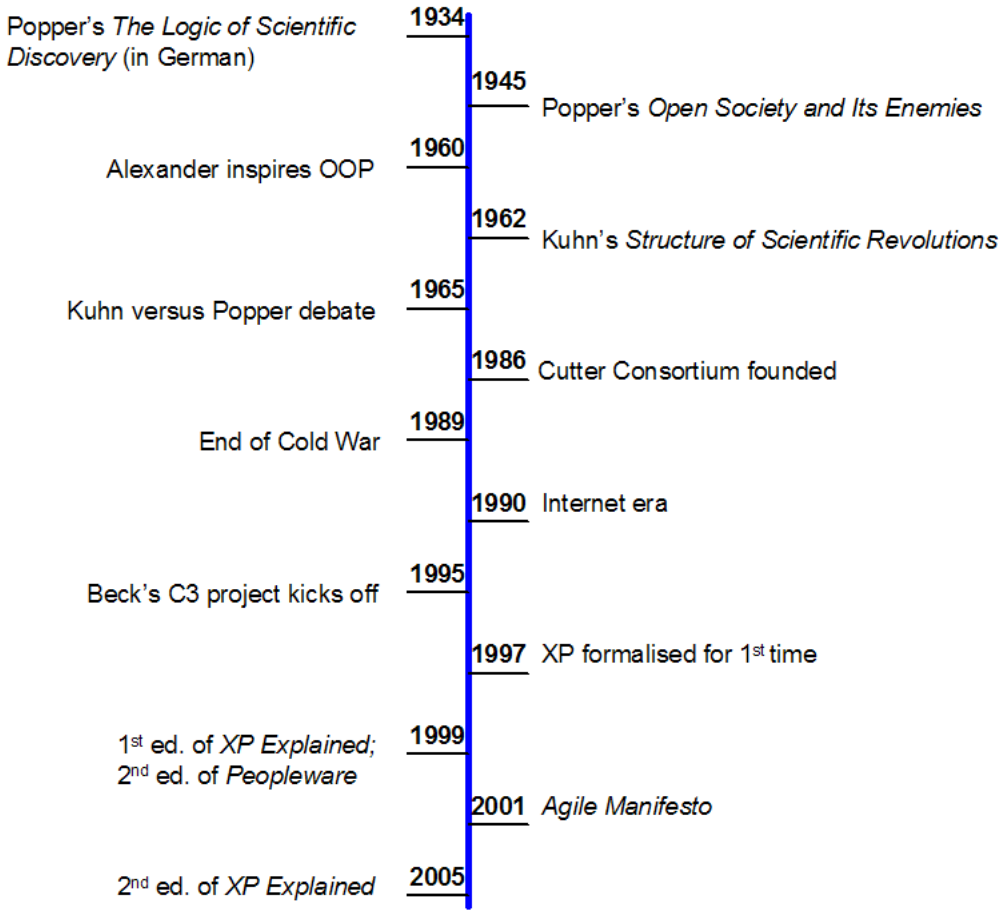


Figure 2.4: Combined Timeline of Historically Important Events for this Dissertation

in the "Planning Game". The team sits together and defines *stories* (roughly equivalent to features), which are captured, complete with test cases, on story cards. The story cards contain hand-written text in the customer's own words. In order to make accurate estimates of the time needed to implement each story, pairs of developers often implement several different prototypes of each story before estimating it. These prototypes are referred to as *spikes*. If it is discovered that a story is too large, the story will be split up into smaller stories with the aim of keeping each story discrete and *testable*. Once all stories have been estimated, the customer prioritises the stories according to importance, risk or some other pertinent criterion[53]. Then, the developers, together with the customer, determine the release plan, which includes a *mutually beneficial* agreement on the iteration length (usually two weeks), the release length (number of iterations), and the project velocity (the number of reasonably equal-sized stories that can fit into an iteration). The planning of

[53]This approach underlies the XP principle of *economics*.

iterations is driven by fixed time intervals (and usually fixed costs and quality too), whereas the scope (the precise stories and how many will be included per iteration) is continuously negotiated. This is what XP calls *negotiated scope contract*[54], an approach that is quite different to the approach of traditional project planning in which all variables, especially scope, are fixed a priori.

Once the exploration phase is complete, the *planning phase* (which is usually a day or two in length) commences. The aim of this phase is to plan the iterations. The stories for the first iteration are usually chosen with the system's architecture in mind. For all subsequent iterations, the customer is free to choose stories based on both the priorities of the remaining stories and on the project velocity. Since the priorities may change throughout the project life cycle, the start of a new iteration gives the customer the opportunity to reprioritise stories, modify existing stories or introduce new stories. The start of a new iteration also gives the developers the opportunity to include new technologies and techniques into the project. In this way, XP is able to easily incorporate changing requirements and technologies into the project throughout the life cycle, making it flexible and open to change. XP also recommends that a few minor stories be included in each iteration, which can be discarded in case the team falls behind schedule.

At the start of each iteration, developers select the stories they would like to be *responsible* for implementing during the iteration. All the stories for an iteration are posted on bulletin boards or whiteboards, which form part of the *informative development workspace*. Workstations in the workspace are arranged to support *pair programming* and other forms of informal collaboration, thereby facilitating *feedback* and encouraging *communication*. The workspace is open and large enough for the *whole team* (including programmers, testers, designers, architects, project and product managers, executives, technical writers, and users) to *sit together*.[55]

The team begins each day with a 10-15 minute stand-up meeting during which team members report on what they accomplished the previous day, raise any problems they encountered, and discuss the task(s) (part of a story) they plan to complete on that day. During these sessions, team members are encouraged to speak truths, pleasant or unpleasant, thereby fostering trust as well as *courage*. Team members are also expected to *respect*

[54] Beck & Andres (2005) (p. 69)

[55] Some of these features are also clearly relevant to *systems theory*, in particular the emphasis on feedback and on the *whole* instead of the *parts*. The awareness of subsystems is also a common theme.

the contributions and suggestions that each team member makes, thereby acknowledging each other's *humanity* as well as the *diversity* of the team. At the end of the stand-up meeting, team members who share common tasks, pair up for *pair programming* sessions. It is not uncommon for pairs to rotate every few hours rather than to work together the whole day. At the start of a pair programming session, developers begin by writing unit test cases for their task and then set about incrementally implementing the code needed to ensure all test cases pass by the end of the session. This approach towards development, called *test-first programming*, is in stark contrast to the traditional approach towards software development in which testing occurs at the end of a project, once all the functionality is already implemented. If a pair is not sure how to proceed with a task, a quick design session is held with a few stakeholders (which often includes the *on-site customer*) in order to work through the problem either by discussing it or modeling it on whiteboards. Often, a pair will identify that in order to continue with a task, it is necessary to *refactor* part of the code in order to keep its design *simple* without changing the underlying functionality. Refactoring the code then becomes the task that the pair works on during the session (SDA, 2003).

At the end of each pair programming session, pairs integrate their test cases as well as the changes they have made to the *single code base* into the *shared code* repository. The aim of this *continuous integration* is to ensure that the most recent changes are included into the automated *ten-minute build* of the whole system. It also aims to ensure quick, clear feedback cycles, which reduce the high cost and unpredictability of delayed integration.

Each work day lasts not much longer than eight hours, a commitment supported by XP's principles of a *forty hour week* and *energised work*. At the end of each day, the whole XP team meets up in a communal area to *reflect* on the day. The stories on the bulletin boards are updated based on the day's progress. This daily process is repeated until the end of an iteration is reached. Then, customers run their acceptance tests and if the results are satisfactory the next iteration begins. If the results of the tests are not satisfactory, the next iteration will involve rectifying the problems, with a focus on *continuous improvement* and an attitude towards viewing problems or *failures* as *opportunities for change*.

Once all iterations for a release are complete, the *production phase* is entered. Developers produce a version of the production-quality software, which can be, but need not necessarily be, released to the user community. This approach of *incremental deployment* ensures that

a continuous *flow* of valuable software in small increments is released to the user community, which, in turn, encourages feedback. Then the planning for the next release begins and the *maintenance phase* for the previous releases ensues. Figure 2.5 diagrammatically depicts the entire XP life cycle, as described in this section.

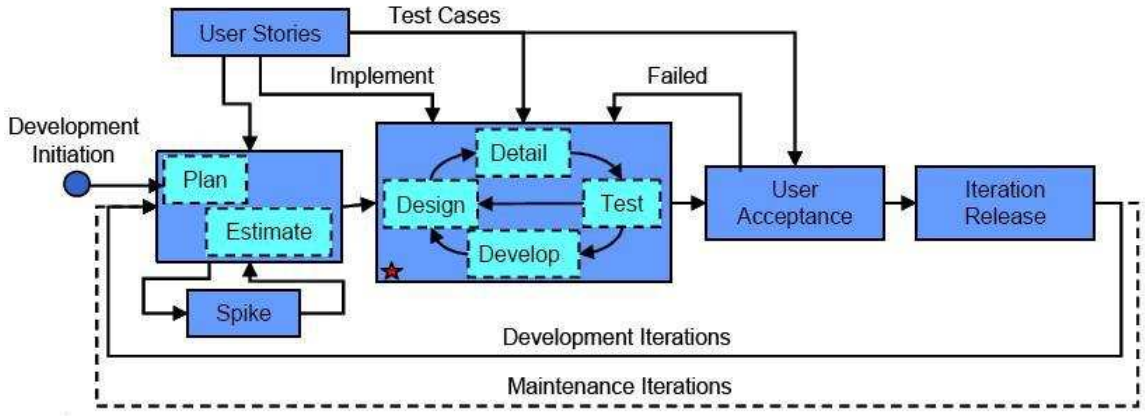


Figure 2.5: The XP Life Cycle (Norton & Murphy, 2007)

From the above description of a typical iteration in the life cycle of an XP project, a few central principles of the XP approach towards software development (which arguably make it revolutionary in comparison to traditional methodologies) have become clear, including XP's:

1. *iterative-incremental approach* towards software development, which encourages continuous feedback and ensures software is built in small increments;

2. emphasis on *stringent testing* through the practices of test-first programming, unit testing and ten-minute builds;

3. emphasis on *open communication* through collaborative practices like pair programming, real customer involvement and story negotiation;

4. aim of a *democratic and humane workspace* in which the whole XP team sits together and collaborates, respecting each others contributions; and

5. flexibility to adapt easily to *changing circumstances and requirements* since software is developed in short iterations.

In section 6.1.2, these principles of the XP approach towards software development will be contrasted with those underlying traditional software methodologies, like the *Waterfall*

Model, which are supposedly less flexible and, which place less emphasis on humanity and individuality than the Agile methodologies.

Before concluding this chapter, the results of a few empirical studies, which have been conducted into the impacts of XP in a real-world setting, will be provided. Williams *et al.* (2004) "provides a benchmark measurement framework for researchers and practitioners to express concretely the XP practices the organization has selected to adopt and/or modify, and the outcome thereof". The authors claim that "[s]oftware organizations are progressively adopting the development practices associated with the Extreme Programming (XP) methodology" but that "[m]ost reports on the efficacy of these practices are anecdotal". In their article, the authors present the initial validation of their XP evaluation framework based on a year-long study of an IBM team that adopted a subset of the XP practices. Layman *et al.* (2004) conducted a case study at *Sabre Airline Solutions* to evaluate the effects of adopting XP practices with a team that had characteristically plan-driven risk factors. The authors created the *Extreme Programming Evaluation Framework (XP-EF)*, as a means for structuring their metrics collection. The case study compares two releases of the same product: "One release was completed just prior to the team's adoption of the XP methodology, and the other was completed after approximately two years of XP use. Comparisons of the new release project results to the old release project results show a 50% increase in productivity, a 65% improvement in pre-release quality, and a 35% improvement in post-release quality". According to the authors, their findings suggest "that, over time, adopting the XP process can result in increased productivity and quality". McDowell *et al.* (2002) report the results of a controlled study designed to measure the effects of pair programming on the development of individual programming ability. Their findings indicate "significant improvements in individual programming skill for students with lower SAT[56] scores". Additionally, they found "that all students are more likely to complete the course successfully when using pair programming".

[56]The Scholastic Assessment Test (SAT), which is often taken by high school juniors and seniors in the USA as a precursor to university admission, assesses a student's verbal, mathematical, and writing skills.

Chapter 3

Related Work

This chapter[1] will provide an account of the literature containing work related to the topic of this dissertation. Work is considered related if it explicitly or implicitly uses the ideas of *professional* philosophers to illuminate aspects of the software engineering discipline. Excluded from this chapter is literature which uses the word "philosophy" loosely to mean "guiding principle" or "general outlook on life"[2]. The related work in this chapter, will contain both a general account of the literature, which uses a broad range of philosophies in relation to the software engineering discipline, as well as a more detailed account of the literature, which uses the specific ideas of Popper and Kuhn within the discipline. These accounts will provide a comprehensive overview of the ways in which philosophy has been used to date within the software engineering discipline, thereby highlighting areas of the discipline which have not yet been the subject of philosophical discussions or investigations.

In order to contextualise the review of the related work in this chapter, the reader's attention is drawn to the fact that much recent debate within the field of software engineering concerns the question of whether to use traditional, plan-driven software methodologies or more Agile ones[3]. Members of the traditionalist camp prefer to follow a strictly pre-defined development process, whereas members of the Agile methodology camp—with considerable

[1]Parts of this chapter have been adapted from Northover *et al.* (2007a) and Northover *et al.* (2008), for which I was the main author.

[2]For an example of the use of the word "philosophy" in this sense, see GNU (2008), which describes the "philosophy" of the *Free Software Movement*.

[3]The distinction between traditional and Agile software methodologies is not meant to imply that all software methodologies can be placed into either the one or the other methodology "camp". Instead, the distinction is made in the hope of illuminating Agile software methodologies, which, as the previous chapter described, emerged historically in response to the failure of traditional methodologies to cope with rapid change during the Internet era.

sympathy for the ideas of Feyerabend[4]—prefer to react to events and circumstances as they emerge. From a social-philosophical viewpoint, it is interesting to note that membership to the one or the other camp is not only a question of knowledge or reasonable insight, but also a question of authority and power.[5] It is also interesting to note that even in the field of classical mechanical engineering—long before software engineering came into existence—there were already examples of Agile approaches. Classical engineering, however, termed these approaches *concurrent* engineering (Smith, 1997).

3.1 Miscellaneous Philosophers and Software Engineering

This section will summarise the literature which uses certain of the ideas of *Plato*, *Aristotle*, *Lakatos*, *Feyerabend*, *Dooyeweerd*, *Wittgenstein*, *Habermas* and *Himanen* to describe a range of aspects within the software engineering discipline.

Lakatos

Marick (2004) argues that the decision to discard an existing software methodology in favour of a new one requires a "gestalt switch" from one ontology to another. In particular, the author discusses the Agile ontology and turns to science in an attempt to understand why many software engineers have adopted this particular ontology. He conjectures that Kuhn's concept of "paradigm shift"—where a paradigm in software terms is presumed to be parallel to a software methodology—may provide one possible explanation: "The [new] paradigm is acquired through practice... The old paradigm dies off with its adherents". However, the author argues that the theory of another 20^{th} century philosopher, *Lakatos*, may provide an even better understanding of this switch than Kuhn's or even Popper's ideas would. He describes the four central aspects of Lakatos' methodology of *scientific research programmes*—a hard core at the heart of each research programme; the importance of novel, striking successes; the progressiveness of research programmes; and the resolute ignoring of counter examples—and finds each of these aspects to be evident in the Agile ontology, in

[4]For a brief overview of Feyerabendianism in software engineering see Gruner (2007). For a critique of Feyerabendianism in software engineering see Snelting (1997). An example of Feyerabendianism in software engineering can be found on the Internet at http://www.dreamsongs.com/Feyerabend/ETAPS03/.

[5]A colleague of mine in the Espresso Research Group at the University of Pretoria, Morkel Theunissen, once mentioned the powerful role of the American DARPA doctrine in the widespread adoption of the strictly hierarchy-procedural development process both in classical mechanical engineering, and software engineering.

some form. The paper concludes with the statement that there is no "Platonic Right way to build software"[6] and that for a methodology to be progressive, one must find the correct ontology.

Plato

Giguette (2006) provides a compelling argument for a *Platonic* view of objects in an object-oriented programming (OOP)[7] language. The class to which an object belongs is compared "to a platonic form, because it generalises all possible class instances."[8]

Aristotle

Rayside & Campbell (2000) claim that "the object-oriented community at times maintains that object-oriented programming has drawn inspiration from philosophy, specifically that of Aristotle". One of the main aims of their paper is to "explain aspects of Aristotelian logic to the computer science research community". Their paper goes into great detail applying Aristotle's system of syllogistic logic to OOP and points out the similarity, on the one hand, between Aristotle's concept of *matter* (which defines the singular or the individual) and OOP's notion of *object* and, on the other hand, Aristotle's concept of *form* (which defines the universal) and OOP's notion of *class*. The authors conclude the paper by stating:

> It has been our contention that Aristotelian logic will be of benefit to programmers in performing their art *with order, with ease and without error.*

Aristotle's concepts of *matter* and *form* are also used by Brooks (1987), a well-known software engineer and computer scientist. He explicitly acknowledges Aristotle and uses the Aristotelian terms *essence* and *accidents* in distinguishing the essential difficulties—those inherent in the nature of software, like complexity—from the accidental difficulties—those that attend its production but are not inherent, like inadequate tools and methods.

[6] Marick (2004) (p. 71)

[7] Object-oriented programming is a style of programming which uses classes, objects and methods to design computer applications. Classes define the abstract characteristics of an object, including its attributes and behaviours. An object is a particular instance of a class and its methods define the object's abilities. For example, the class Bird defines all possible birds and the object Dove is an instance of the Bird class. Birds have the ability to fly, which is one of Dove's methods.

[8] In this author's opinion, OOP would be better characterized as Aristotelian rather than Platonic, but this discussion is outside the scope of this dissertation.

Himanen

The Finnish philosopher Himanen extensively uses the ideas of Plato and Aristotle in relation to Open Source Software Development (OSSD). He argues that OSSD is the modern day descendant of the *hacker ethic*, which he favours strongly over the more widespread, *protestant work ethic*, which is based on a capitalist mindset of work-centeredness. Himanen (2001) argues that the hacker ethic is closely related to the virtue ethics found in the writings of Plato and Aristotle. In the first part of the book, Himanen argues that the hacker **work** ethic, which is based on passion, joy, creativity and interest, is similar to the attitude of passionate intellectual inquiry expressed by Plato nearly 2500 years ago in the *Symposium* and the *Phaedrus*. Himanen also points out the underlying similarity between the hacker work ethic and Plato's emphasis on the literal meaning of the word *philosophy*, namely, "a passion or love for wisdom". Regarding the Protestant attitude towards time—that "time is money"—Himanen argues that this attitude gives rise to a 'culture of speed' and to an overly rigid *optimisation* and *organization* of time. This attitude towards time, he argues, is not supported by the hacker ethic since hackers have the freedom to organise their own time. This attitude is similar to the way Plato defined the academic relation to time: a free person has *skhole*, that is "plenty of time". The Protestant attitude towards time, Himanen argues, affects creativity and means that people are no longer in charge of their own time, a symptom which Plato called *askholia* and associated with a state of imprisonment or slavery. In the second part of his book, Himanen contrasts what he calls the Protestant *money* ethic—that "the earning of money and more money is the highest good" and that both work and money are ends in themselves—with the hacker ethic's relation to money, namely that *peer recognition* as opposed to making money is the main motivation. He points out that in our society, which is infused by the Protestant work ethic, work is actually a source of social acceptance. He contrasts this state of affairs with the ideal societies Aristotle described in his *Politics*, in which "only those who did not have to work were considered worthy of citizenship". Furthermore, he argues that, in contrast to the Protestant money ethic, which leads to the idea of *ownership* through patents, copyrights and other means, the hacker ethic emphasises openness and the free flow of information. It has its historical roots in the idea that scientific knowledge must be public. This is an idea which, according to Himanen, had its roots in the ethic of the first scientific community, Plato's Academy, which was based on *synusia*—concerted action in which knowledge was

shared freely—and on Plato's idea that "no free person should learn anything like a slave". Related to the openness and free flow of information, is the equally important idea in the scientific ethic of skepticism and critical enquiry, which, Himanen argues, is a continuation of the *synusia* of Plato's Academy. The scientific ethic as well as the method of deductive problem solving is central to the open-source model, according to Himanen. A final resemblance between the hacker ethic and Plato's Academy, is the experts' common regard for students as companions in learning as opposed to targets for knowledge transmission, and an understanding of themselves as learners who act as gadflies, thanks to their deeper knowledge.

Dooyeweerd

An authority in applying *Dooyeweerd's* philosophy to Information Systems, Basden, has recently published a monograph entitled "Philosophical Frameworks for Understanding Information Systems" (Basden, 2008). In this book, he aims to develop a philosophical framework based on Dooyeweerd's philosophy for understanding the five main areas of research and practice in Information Systems: the nature of computers and information, the creation of information technologies, the development of artefacts for human use, the usage of information systems, and information technology as our environment.

Feyerabend

Snelting (1997) points out "that in software technology sometimes things happen quite the way *Feyerabend* ... describe[s] them". In particular, he notes that empirical studies are rare in computer science (by investigating academic papers which appeared in conferences and journals for evidence of experiments), that there exists a chasm between theory and practice, and that a particular form of *constructivism* is rife amongst "practical scientists, who ignore theory, and at the same time avoid empirical validation". He argues for "a stronger empirical foundation for software technology". To achieve this, he insists basic scientific principles should not be neglected, for instance, predictions should always be falsifiable. Regarding this particular form of (social) constructivism, the author points out that computer science partially constructs its own reality since computer scientists invent abstract concepts or devices, for example, abstract data types, software architectures, and design patterns. For this reason, "it is popular to call computer science ... [a] 'structural

science', however recently computer science is seen more as an engineering science". The author goes on to question "What is the computer science equivalent of predictions and falsifying experiments?" and points out that predictions in computer science are obtained from abstractions and theories. More recently, he argues, there have also been methods of prediction, such as the application of model checking[9]. Experiments, on the other hand, are most importantly found in the form of software testing, although "testing, like all experiments, can only be used to refute a specific prediction". The author concludes that "the fundamental science-theoretic concept of falsifiability [*sic*] is as valid in software technology as it is in the natural sciences, but is often ignored". Finally, he argues that the high standards of the natural sciences should be applied not only to computer science but also to academic research in this field.

Habermas

Lyytinen & Klein (1985) observe that research into how information systems are used and developed are traditionally based on an engineering model, which is limited insofar as it focuses predominantly on the technical aspects of design at the expense of the social aspects. Therefore, they've put forward a theory, derived from Habermas' Critical Social Theory[10], which views design of information systems as a social process.

Wittgenstein

Cockburn (2002a) points out how Ehn (1988) considers software development in the context of four philosophers, namely, *Descartes*, *Marx*, *Heidegger*, and *Wittgenstein*. Ehn is said to go into great detail applying Wittgenstein's *Language Games* to software development, which is seen as a cooperative game of invention and communication. Using the style of Wittgenstein, software design is seen as the unfolding of a language game, in which new words are added to the language over time.

[9]Model checking is a method to check a finite-state machine against a formal specification and to generate counterexamples or refutations if the specification is violated.

[10]Critical Social Theory is a school of thought, which was formed in the 1930s by a group of students (originally Horkheimer, Adorno, Marcuse and Fromm) affiliated with the Institute of Social Research (which is more commonly known nowadays by the informal term the 'Frankfurt School') at the University of Frankfurt. Their main criticism was of scientism since it deals narrowly with 'Instrumental Reason' without considering social theory.

Summary

The review of the related work in this section has shown that several software engineers have used the ideas of professional philosophers to help understand their discipline. It was shown that the ideas of a wide range of philosophers, from the Ancient Greeks—Plato and Aristotle—to the contemporary 21^{st} century philosophers, Lakatos and Feyerabend—have been used to describe diverse aspects of the software engineering discipline. In some cases, the ideas of different philosophers have been used to describe the same aspect of software engineering—for instance, both Plato and Aristotle were used in relation to object-oriented programming. In other cases, the ideas of the same philosopher were used to describe several seemingly unrelated aspects in the discipline—for instance, Plato's philosophy was used in relation to both object-oriented programming and open source software development. The following two sections will focus on the literature, which specifically uses the respective ideas of Kuhn and Popper within the software engineering discipline.

3.2 Kuhn's Ideas in Software Engineering

Kuhn's concepts of *paradigm shift* and *scientific revolution* have been used, explicitly and implicitly, to describe diverse aspects of the software engineering discipline, most notably by certain members of the *Cutter Consortium*[11]—an international IT advisory firm dealing with several core business areas, including "Agile project management". These specific uses of Kuhnian concepts by members of the Cutter Consortium are of particular significance to this dissertation since several leaders of both the Agile project management and software development movements are amongst the members of this Consortium[12] who have used Kuhn's ideas in their own work on Agile methodologies. For this reason, most of the work summarised in this section has been taken from the literature of these authors, whose writings are published in the Cutter Consortium's own book series.[13] Before summarising the related work of these authors, a few particularly important references from authors who are not affiliated with the Cutter Consortium, will first be summarised. The latter

[11] CCO (2008)

[12] These members include Kent Beck, Alistair Cockburn, Ward Cunningham, Jim Highsmith, Ron Jeffries and Ken Schwaber, amongst others.

[13] These writings can be found in the Cutter Consortium's online bookstore at https://cutter.com/cgi-bin/catalog/store.cgi. Note, however, that the full-text versions are only accessible to subscribing members.

authors have explicitly and consciously drawn analogies between Kuhn's account of scientific paradigms and the current state of software engineering.

In their paper entitled *Can Thomas Kuhn's paradigms help us understand software engineering?*, Wernick & Hall (2004) state that they based their research into this question on both *theory* (by analysing software engineering textbooks) and *practice* (by interviewing practitioners). In particular, they drew "an analogy between Kuhn's account of scientific disciplines, and the current state of software engineering", and used Kuhn's concept of a *paradigm* or *Disciplinary Matrix (DM)* to analyse the mindset induced by the use of predefined mechanisms in software engineering like notations, tools and methodologies, which, they argue, impact on the result of the software engineering process. The authors conjectured that their model, drawn by analogy from the philosophy of science, "may provide the basis for improved software engineering education, by enabling the teaching of a deeper understanding of current and past fundamental underlying principles" and that their model "works well both in describing the current state of software engineering and in providing new ways of approaching its perceived problems". The authors point out that they are not the first to consider software engineering from a philosophical perspective using science as an analogy:

> Hirschheim and Klein's (1989) 'functionalist paradigm' underlies methodologies such as Structured Analysis and Information Engineering; in their view, many successful software system developments attempt "...to follow the scientific method". Wood-Harper and Fitzgerald (1982) refer to two different types of approach to software development, which are designated 'science' and 'systems' and used to classify a set of six lower-level approaches.

The authors justify their specific use of Kuhn's philosophy using similar arguments to those that the reader will find in chapter 4 of this dissertation[14]:

- The similarity between the current status of software engineering and Kuhn's 'pre-paradigm' stage of scientific development;
- The parallel between Kuhnian 'normal science' and applying software engineering tools and techniques to develop a software product;

[14]Note that these arguments were discovered independently and at different periods of time in history. This is an important point to bear in mind for section 5.3, which discusses Popper's three worlds metaphysical model. The fact that the arguments were discovered independently, Popper would argue, bears testimony to the existence of his objective World Three of knowledge.

- Curiosity as to whether the frequent claims of 'paradigm shifts' in the practice of software development, most frequently in sales literature..., can be justified by reference to Kuhn's original formulation of the term.

They conclude the first half of their paper by stating: "The results of this study suggest that the current status of the theory of software engineering parallels Kuhn's 'pre-paradigm' stage of scientific development." and "that the application by analogy of Kuhn's view of scientific activity to software engineering is justifiable".

In the second half of their paper, the authors describe their fieldwork and results. However, these results seem mostly unfalsifiable. On the one hand, the authors state that they considered fieldwork participants to be operating within the same "paradigm" if, in the case studies, they shared similar beliefs—like scientists practising Kuhnian *normal science*. On the other hand, if their beliefs differed, the authors considered the participants to be operating within a Kuhnian *pre-scientific* paradigm due to their lack of consensus. Therefore, it seems that the authors looked for ways to justify their categorisation of the participants in terms of Kuhn's philosophy by subjectively choosing the criteria for such categorisation. It can also be questioned whether Kuhn's concept of a "paradigm" can justifiably be applied to software engineering *as a whole*, as these authors have attempted to do.

The second important reference to Kuhn from outside the Cutter Consortium is made by Castells, a sociologist whose research into *information society* and *communications* has resulted in him being the fourth most cited social sciences scholar and the foremost cited communications scholar in the world. In the epilogue of Castells (2001), he explicitly uses Kuhn's notions of "paradigm" and "disciplinary matrix" to argue that the new technological paradigm of *informationalism* (which, he believes, was partly invented and decisively shaped by events such as the development of computer networking and the distribution of computer processing capacity), "is currently replacing industrialism as the dominant matrix of twenty-first century societies". In particular, Castells writes regarding Kuhn:

> Technological systems evolve gradually until a major qualitative change occurs: a technological revolution, ushering in a new technological paradigm[15]. The

[15]The reader's attention is drawn to Castells' use of the words "evolve" and "revolution", which are strong themes throughout this dissertation due to their association with the respective philosophies of Popper and Kuhn. The reader's attention is also drawn to the fact that, in Appendix F, several important ethical implications and challenges that the technological revolution raises, will be discussed in relation to the philosophies of Kuhn and Popper.

> notion of paradigm was first proposed by the leading historian of science Thomas Kuhn to explain the transformations of knowledge by scientific revolutions.[16]

Castells raises the question "How do we know a given paradigm (e.g. informationalism) is dominant vis-à-vis others (e.g. industrialism)?" and argues that it simply comes down to "its superior performance in the accumulation of wealth and power". This observation highlights the fact that the choice between paradigms is not usually a matter of logic and reasoning but rather a matter of authority and power. As will be shown later in this dissertation, Kuhn would agree with Castell's position, whereas, the anti-authoritarian and critical rationalist ideal Popper's advocates, would be strongly opposed to it.

The final reference that will be considered in this review of the related work from outside the Cutter Consortium, is made by Constantine (2001). The author uses Kuhnian terminology throughout his book, although he seems to do so unconsciously, since he neither cites Kuhn's work, nor mentions Kuhn explicitly. He speaks of object-orientation as a revolution which overthrew the previously dominant paradigm of procedural programming, and argues that:

> many of the early pedagogues and demagogues of OO preached that the only route to objective effectiveness was to give up the ways of the past, to forget everything... and learn a whole new paradigm. What they advocated had more the ring of religious conversion than of learning new technical skills and concepts.... Get them while they're young and untainted by procedures and you can make them true believers.

Constantine's statement not only contains references to Kuhnian concepts like "paradigm" and "revolution", but also contains many Kuhnian ideas like indoctrinating young scientists, rewriting textbooks and, most importantly, discarding the previous paradigm in its entirety before holistically adopting the new paradigm. All of these ideas will be elaborated on in section 4 of this dissertation.[17]

[16] Castells (2001) (p. 155)

[17] I have personally heard of an account where, after 'converting' to object-orientation, designing in any other way was not possible, since the OO ideas had taken root, and they dominated any other way of thinking. On the other hand, I've heard of cases where older programmers could not take to OO, however hard they tried. These accounts seem to corroborate Kuhn's belief in the incommensurability of different paradigms.

Summary

This summary of the related work of three authors outside the Cutter Consortium, showed that the first two authors explicitly used Kuhn's ideas to understand the current *state* of the software engineering discipline, rather than to explain a particular aspect of the discipline. The third author, on the other hand, used Kuhnian ideas more narrowly to describe a particular aspect of the discipline, namely, the new 'paradigm' of object-orientation. Therefore, it seems that those authors who have used Kuhn's ideas explicitly, have done so at a broader level: they have attempted to explain the software engineering discipline as a whole rather than merely an aspect of it. This dissertation conjectures that Kuhn's ideas better explains aspects of the software engineering discipline when applied at a broader level, rather than at a more detailed level. The remainder of this section will summarise the uses of Kuhn's ideas by several members of the Cutter Consortium, in their literature relating to Agile software methodologies.

Cutter Consortium

The most notable of all references to Kuhn, for the purposes of this dissertation, by a member of the Cutter Consortium, is undoubtedly that made by Beck, the founder of Extreme Programming (XP)—arguably the most prominent Agile methodology. He explicitly cites Kuhn's *The Structure of Scientific Revolutions* in the annotated bibliography of both editions of his own book, *Extreme Programming Explained: Embrace Change*[18]. In addition to these explicit citations, in a personal e-mail communication between myself and Beck in 2006, he acknowledged that Kuhn's book was influential on his thinking.

Beck (2002) uses Kuhnian terminology in pointing out, in a section entitled "Viva la Revolución", that:

> When a group of us met in Snowbird, Utah, in February 2001 and issued the Agile Manifesto, we understood that agile is a revolution. That is why the manifesto, a call to arms, was based upon principles rather than techniques. Agile process management represents a profound shift in the development of products and software.[19]

[18] Beck & Andres (2000) and Beck & Andres (2005)

[19] It should be noted that Beck is using the term "revolution" here in a political sense rather than in Kuhn's scientific sense. Perhaps he is comparing the Agile leaders to the rebels of the American War of

Bach applies the work of Popper, Kuhn and several other philosophers of science to software engineering in the context of the *process versus practice* debate. He argues that in the 20^{th} century, even science was a battleground for this debate when the idea of science as a rational enterprise came under fire. The resulting fallible view of science, according to Bach, is pertinent to our situation in information technology today. Bach (2000) also argues, perhaps controversially, that:

> ...developing software is not much different from developing scientific theory, and developing processes for developing software is exactly like doing science.

Furthermore, Bach believes that the trend to move from formalised software development processes towards more intuitive practices manifests itself not only in Agile methodologies but also in the object-oriented design patterns approach.[20]

Although Schwaber, another Cutter member, does not explicitly cite Kuhn, he frequently uses the Kuhnian term "revolution" in his writings. Schwaber (2001) describes the seminal meeting of the Agile advocates in 2001 as a "meeting of revolutionaries", the main objective of which, according to Schwaber, was "to discuss revolution":

> You cannot imagine the meeting of the revolutionaries. Such passion, such fire, such depth of experience...

Marzolf & Guttman (2002) use Kuhn's theory to explain the movement known as *systems thinking* (ST) and how it relates to software engineering. They point out that, although systems thinking requires systems to be addressed holistically, the most universal characteristic of software development is, rather, a short-term, piecemeal approach:

> This stark contrast...is at the heart of the current controversy between the so-called defined and agile development approaches.

The authors go on to ask: "Should we focus more single-mindedly on building software one application at a time, or should we take a more holistic view?"[21] and conclude that

Independence (pp. 1776-1783). However, the "American Revolution" was merely a change of the ruling class, rather than a complete social, economic and political upheaval as was the case in both the French Revolution (p. 1789) and the Bolshevik Revolution (p. 1917). Therefore, in this case, the term *rebellion* may be more appropriate than *revolution*.

[20] The latter approach was fundamentally influenced by the adoption of the architect, Christopher Alexander's, ideas on housing architecture and civil engineering (Alexander, 1999) into the domain of software engineering. Alexander's ideas, and specifically his influence on Agile software methodologies, will be discussed in detail in section 7.1.

[21] This holistic approach could be considered Kuhnian, as will be shown in chapter 4, just as the piecemeal approach could be considered Popperian, as will be shown in chapter 5.

software engineers should learn ST and take a more holistic approach towards building software. This, in their opinion, will ensure the shift from the "machine age paradigm" of the 19^{th} to the contemporary "systems age paradigm".

Mah (2004) talks about *revolutions* and *paradigm shifts* in relation to celestial spheres. The author mentions Copernicus, Galileo, Kepler and Newton, and posits that:

> Our modern-day Copernicuses and Galileos are folks like De-Marco, Lister, Putnam, Yourdon, Orr, et al.[22]

Furthermore, using terminology similar to that used by Marzolf and Guttman, Mah argues that software engineers still seem to be operating in the *Industrial Age Paradigm* instead of the *Information Age Paradigm* when they talk about "manufacturing"—as opposed to "designing"—software systems. In Mah's writings, one can thus observe how the similarity between classical mechanical engineering and software engineering is both recognized and rejected.

Windholtz (2002) paraphrases several talks from the 2002 XP Agile Universe conference. Most notable, for the purpose of this dissertation, are the talks by West and Fowler. At the conference, West attempted to answer, firstly, why there is an undercurrent of revolution in the Agile community and, secondly, how one should properly conduct a revolution. Similarly, Fowler argued that the Agile and XP movements are a large-scale shift in how people think about software development:

> The change in attitude is twofold: firstly a change in attitude toward change and secondly a change in attitude toward people.

Davies (2006) explicitly cites Kuhn's *Structure of Scientific Revolutions* in a conference address and explains, with reference to Wittgenstein's Duck-Rabbit[23], that understanding Agile Software Development requires a *paradigm shift*.

Yourdon (2001b), like Davies, references Kuhn's *Structure* explicitly and applies Kuhn's concept of "paradigm shift" to information technology organisations in the context of ad-hoc communication networks. Yourdon concludes his article by stating:

[22] By linking the latter authors from the software engineering community to revolutionaries in science like Newton and Galileo, the author seems to be suggesting that the ideas of these authors in relation to Agile methodologies can also be considered revolutionary. If this is the case, then Kuhn's ideas may well be useful in explaining aspects of the Agile methodology "revolution".

[23] The Duck-Rabbit, made famous by Wittgenstein, is a picture which can be seen in two distinct ways, as either a duck or a rabbit. Wittgenstein uses this picture in his *Philosophical Investigations* as an aid to an examination of what is meant by "seeing as".

> No doubt there will be other paradigm shifts, too, including some that the current generation of IT managers and professionals will resist strongly. This is not an unusual situation: Kuhn's Structure goes to great pains to explain why and how the defenders of old paradigms battle the revolutionaries bearing new paradigms.

Yourdon (2001d), in a related paper, applies Kuhnian terminology specifically to XP by describing "classical" software engineering as the *old paradigm* and "agile", "lean" or "light" software development as the *new paradigm*. Yourdon goes on to point out that Beck himself explicitly described Extreme Programming as a "paradigm shift" in his presentation at the 2001 Cutter Summit Conference. Furthermore, Yourdon raises the critical question whether exceptions and special cases in the XP paradigm may soon occur, which would ultimately result in its replacement by yet another new paradigm. Yourdon (2001a) likens the *new paradigm* to open-source software rather than Agile software, and Yourdon (2001c) debates whether peer-to-peer computing will be the next big *paradigm shift*.

Welsh (2001) applies Kuhn's theory to Model Driven Architecture (MDA). The author argues that in order to win over a generation of developers to MDA (who are comfortable with third generation programming languages like Java), a *paradigm shift* will be required.

Puccinelli *et al.* (2000) cite both Michael Polanyi's *Personal Knowledge* and Kuhn's *Structure* while attempting to argue against the objectivity of knowledge in the context of Organisational Knowledge Management (OKM).

Finally, Bloomberg (2002) argues that a new *paradigm* for IT has been enabled by *Web Services*.

From the account of the related work above, it is evident that Kuhn's ideas have been used by several members of the Cutter Consortium in relation to Agile software methodologies. However, it is conjectured that many of the authors have used Kuhn's ideas uncritically and superficially, without having a comprehensive understanding or knowledge of Kuhn's philosophy. It appears that the authors may be taking advantage of the hype of the Kuhnian buzzwords "paradigm shift" and "revolution", perhaps to further their own agenda, namely, to popularise and encourage the adoption of Agile methodologies. Consequently, one of the central aims of this dissertation is to provide a seemingly much needed critical perspective on the usefulness of Kuhn's ideas in explaining aspects of Agile software methodologies, especially XP. This critique will begin in the following chapter of this dissertation where

attention will be focused specifically on Yourdon's use of Kuhn's ideas since, besides Beck, he seems to be Cutter's most prominent and influential member as far as using Kuhnian ideas in software engineering is concerned.

3.3 Popper's Ideas in Software Engineering

This section will consider the literature which specifically uses Popperian ideas to illuminate aspects of both the software engineering and Knowledge Management (KM) disciplines. Firstly, an account will be given of the literature which applies Popper's principle of *falsificationism* to software testing and design, secondly, it will be shown that Popper's *three worlds metaphysical model* has been applied successfully by certain practitioners as a KM framework, and finally, it will be shown that certain software engineers have pointed out the similarities between *open source software* and Popper's *open society*.

3.3.1 Popper's Falsificationism and Software Testing and Design

Bossavit (2008) is the only author found, who uses Popper's ideas in relation to Agile software methodologies. The author proposes a Popperian view of the software design process by arguing that Popper's belief that "[o]n the scientific level, we systematically try to eliminate our false theories—we try to let our false theories die in our stead"[24], stresses the goal of good design: adaptation to a given environment. Furthermore, the author argues that "Popperian design eliminates. It kills hypotheses. So we should be able to verify that a Popperian design process is in operation by finding the remains of discarded theories, hypotheses, or designs... As a corollary, design documents should document not just the current design, but the list of design alternatives which were found to be likely to cause the project to "die", if they had been implemented". The author highlights that a crucial property of a Popperian design process is the ability to make predictions as to how a design alternative will turn out if it is implemented. There are several ways of making such predictions, according to the author, including using a modeling environment, or "checking for conformity to design patterns or the Law of Demeter". Finally, in relation to the author's belief that "the effort involved in eliminating a design alternative must be less than the effort involved in actually realizing that design alternative", the author claims

[24]Magee (1986) (p. 73)

that Agile "development can be seen as a critique of Popperian design processes" since Agile methodologists claim that good testing, the factoring of the software in flexible and modular pieces, and good teamwork make it cheaper to try out design hypotheses in actual code.

Coutts (2008) uses many Popperian ideas to trace the similarities between software testing and the scientific method. He states:

> Thus, falsification is just as essential to software development as it is to scientific development. Whereas in science the falsification criteria [*sic*] is often used as a demarcation between science and non-science, in software the falsification criteria [*sic*] demarks [*sic*] the testing discipline from the analysis and development disciplines.

Snelting (1997) observes that the dictum, "Testing can only demonstrate the presence of errors, not their absence", of the famous Dutch computer scientist, Dijkstra, is a special case of Popper's falsifiability principle. Another pioneer of computer science and software engineering, Hoare, who is the founder of *Communicating Sequential Processes (CSP)*, a formal language for describing patterns of interaction in concurrent systems, makes a similar observation in Hoare (2003b). Hoare (2003a) also points out that "Following Karl Popper's criterion of falsification for the meaning of a scientific theory, Roscoe and Brookes concentrated on failures of these tests, with particular attention to the circumstances in which they could deadlock or fail to terminate. This led to the now standard model of CSP, with traces, refusals, and divergences".

Several authors have used Popper's principle of falsificationism in relation to formal specification and verification. Aichernig (2001), for example, argues that falsification can be applied to software development in the following way:

- A formal specification of requirements helps to solve systematically the problem of software development by offering the apparatus of logic.
- The validation of the formal specification as well as the implemented solution only is feasible through falsification—which is testing.
- For being falsifiable (or testable), the requirements description has to be unambiguous and sound—ensured by formal specification and verification techniques.

Another author who uses Popper's ideas in relation to program verification is Meyer (2008). He states: "There is a view of science first proposed by Popper that any true science must contain open and testable claims. Lakatos has shown that this same testability applies to mathematics in the form of methods (approaches) which are tested by evaluating their problem solving success". The author's intention in pointing out these two claims is to encourage software engineering work, which will lead to a scientific test in computer program verification.

Hamlet (2002) provides an account of his "Philosophy of Software Engineering" in a keynote conference address on "Science, Computer Science, Mathematics and Software Development". He motivates his use of philosophy by arguing that "philosophy can influence our work" and substantiates this claim by quoting Popper "We all have our philosophies... and [they] are not worth very much. But [their] impact upon our actions and our lives is often devastating". Hamlet's keynote address begins by analysing the interrelationship between mathematics, science and engineering and concludes that science aims to describe natural laws, that engineering must conform to these natural laws, and that mathematics is the handmaiden of science and the tool of engineering. This is what Hamlet calls the "traditional paradigm", a concept which he attributes explicitly to Kuhn. He also claims, using the Kuhnian idea of "normal science", that much uncertainty and failure is removed when the paradigm is in a "normal" period. Hamlet then uses this "traditional paradigm" as a basis for attempting to answer the question "What is software engineering?". He discusses software engineering's inter-relationship with computer science and logic/algebra. With regard to computer science, he claims that "computer science is not science" and, using Popperian terminology, that "there are no falsifying experiments" in computer science. Instead, according to him, to "experiment in computer science means to implement an idea and force it to work", an approach which is in stark contrast to the physical sciences, since scientists cannot change reality to fit a theory. Furthermore, Hamlet argues that there may be some "science" in computer science since science overlaps with rational discussion. He uses Popper's words "I may be wrong and you may be right, and by an effort we may get nearer to the truth" to substantiate this claim. With regard to logic and algebra, Hamlet discusses software modeling and contends that "Initially, the model changes to fit 'reality'. Later, 'reality' is adjusted to fit the model". Hamlet concludes his keynote address by stating that a fundamental understanding of philosophy should help

one deal with the difficulties of software engineering, like unrealistic time schedules and changing requirements. However, he believes that philosophy will not solve all problems in software engineering since software is not subject to natural laws.

Totten (1999) argues that software should be viewed in terms of a Popperian *conjecture* since software is inherently unreliable. According to the author, software which has withstood severe testing and which seems to function correctly, should be accepted tentatively, always keeping in mind that we can never be sure of it correctness. Over time, many people will hopefully test the software rigorously, and perhaps provide information that will refute it.

Finally, Wernick & Hall (2004) conjectures that "Popper's (1969) concept of falsification can be applied to the testing of software". The focus of that paper, however, was using Kuhn's ideas in relation to software engineering, as was detailed in section 3.2 above.

From the review of the literature in this section, it seems there is some consensus regarding the value of applying Popper's principle of falsificationism to software testing. This section also showed that Popper's principle has been applied by some to program verification. Both aspects will be elaborated on in section 5.1 of this dissertation.

3.3.2 Popper's Three Worlds Metaphysics and Knowledge Management

In this section, it will be shown that Popper's *three worlds metaphysical model* has, according to a few specific practitioners, been successfully used as a Knowledge Management (KM) framework in their organisations. From the outset, it should be noted that the KM discipline is a distinct discipline from the software engineering discipline. However, the two disciplines are not mutually exclusive. For example, the KM discipline is often supported by the products that software engineers create like knowledge bases[25] and document management systems, especially nowadays where the emphasis is on electronic information and "paperless" offices. In turn, organisations which produce software artefacts of any complexity or which have many employees who require access to information resources, usually adopt some type of framework for managing the vast amounts of knowledge in the organisation. Therefore, while the KM discipline is not directly related to the software engineering discipline, an account of the way four specific authors from the KM discipline have used

[25] A knowledge base is a special type of database for knowledge management, which provides the capability for the computerised storage, manipulation and retrieval of information.

Popper's three worlds model will be given below by way of a brief, but relevant, digression. Popper's three worlds model[26], is important to any discipline which deals with knowledge—software engineering included—since it highlights both the fallibility of human knowledge and the importance of maintaining openness in knowledge processing.

Hall (2003a) describes how his organisation successfully applied Popper's three worlds epistemology as a knowledge management framework:

> Tenix Defence, one of Australia's largest defense contractors ... is moving from a paradigm of traditional paper documents to electronically managing and automating structured knowledge artefacts in a knowledge management framework based on Karl Popper's (1973) three worlds of knowledge.

Furthermore, Hall (2003b) claims that Popper's epistemology "provides a much more objective approach [towards] understanding the nature of knowledge [and] provides a much broader epistemological foundation for Organisational Knowledge Management (OKM) than does Polanyi's, and contributes interesting insights into organizational knowledge and memory". Finally, Hall (2005) notes that "only a few authors addressing the KM discipline, including Firestone and McElroy, Blackman et al., Capurro, Gaines, Moss and myself" reference Popper's ideas.

Moss (2003) uses Popper's criterion of falsificationism to examine the scientific status of a number of theories from the fields of knowledge management and organisation theory. The author argues that "many of them suffer from either being phrased in language so vague that they gloss over phenomena or from making predictions that are so cautious and all-encompassing as to be practically useless.[27] As a result, they are likely to be unfalsifiable in Popper's meaning of the term and their epistemological status is called into question". Furthermore, he argues that the vast majority of organisation theory comprises simple statements of general trends such that they lack significant predictive power and are therefore unfalsifiable and epistemologically questionable. Finally, the author concludes that "organisation theory and knowledge management might well benefit from the openness of Popper's approach".

[26] While this model will be discussed in detail in section 5.3, a brief definition of the three worlds may prove useful here. According to Popper, *World 1* is the world of physical objects or of physical states, independent of any perceptions, *World 2* is the subjective world of private mental states, and *World 3* is the objective but intangible world of products of the human mind.

[27] This is in stark contrast to Popper's assertion that hypotheses be bold and precise.

McElroy (2002) argues that "KM has evolved into two distinct bodies of practice: first-generation KM and second-generation KM. First-generation KM is Tayloristic. It begins with the assumption that valuable knowledge already exists. . . Second-generation KM takes a decidedly different stance. Firstly, it is fundamentally Popperian. There is no 'justified true belief' or infallible knowledge, nor does knowledge 'already exist'."

Firestone & McElroy (2003a) argue that KM practices vary according to the different approaches or philosophies businesses have for determining truth. They discuss these competing philosophies of truth and how these influence KM. The authors identify two broad categories of competing philosophies—foundationalist and fallibilist—each containing several categories. They note that while most mainstream businesses are fallibilist, some of those within the fallibilist category—which these authors call justificationists—insist on adopting beliefs and knowledge claims as if they are justifiably true, even though they admit that achieving certainty in knowledge is unachievable. According to the authors, justificationists treat knowledge produced by authorities as though it is *true with certainty*. Consequently, the claims and beliefs of authorities go untested.[28] As an alternative, the authors propose "The New Knowledge Management (TKNM) View" which is based on Popper's critical rationalism. They list, as advantages of this view, the following:

- The quality of knowledge is higher because knowledge is open to criticism.
- The organisations are fundamentally more adaptive because they never view their knowledge as being true with certainty.
- A larger pool of creativity and innovation exists in the organisation because the responsibility of knowledge production is extended to the entire community not just to those in positions of authority. This also leads to higher levels of transparency and inclusiveness.
- Risks are reduced due to the higher quality of knowledge.

The same authors introduce the concept of an "Open Enterprise" in Firestone & McElroy (2003b). They acknowledge that the concept was inspired by Popper's book *The Open Society and Its Enemies* (Popper, 1966) and that it is based on Popper's tetradic schema

[28]The similarity with Kuhn's *normal science* in which scientists in a paradigm are regarded as authorities and their beliefs are not open to criticism should be noted and kept in mind for the discussion in sections 4 and 5.

for problem solving. They also point out that fallibilism and falsificationism are two critical elements that both Popper's open society and the "Open Enterprise" share in common. In formulating their notion of an "Open Enterprise", the authors used the emphasis Popper placed on testing and evaluating knowledge claims (which are inherently uncertain) as well as the emphasis Popper placed on both subjective and objective knowledge. According to the authors, Popper's idea of error elimination is particularly important and, therefore, they use it as "one of the prime distinguishing elements in the Open Enterprise construct". Furthermore, they argue that openness in knowledge processing is a requirement that is paramount to achieving the goal of enhancing *organisational intelligence*. The benefit of the "Open Enterprise", according to McElroy and Firestone, "is the growth of knowledge targeted on solving adaptivity problems, including the growth of knowledge about how to solve problems, innovate, and adapt".

This section has shown that certain practitioners from the KM discipline have deliberately used Popper's ideas—especially his three worlds metaphysical model—as a framework for KM. While these uses of Popper's ideas were not specific to the software engineering discipline, it is conjectured that they can easily be transferred to any discipline which manages knowledge, software engineering included.

3.3.3 Free Open Source Software and Popper's Open Society

This section will summarise the literature which relates Free Open Source Software (FOSS) to Popper's notion of an *open society*. FOSS is characterised by being liberally licensed, which grants users the right to study, change, and improve its design through the availability of its source code. FOSS development (FOSSD) is a widely recognised and continuously growing method of developing software. It differs quite radically from other software development methodologies by emphasising distributed software development, by being less strongly time-bound, and by promoting self-organising teams. The relation between Agile software development—the specific methodology of focus in this dissertation—and FOSSD is often debated. Some believe the two methods to be incompatible while others advocate a hybrid of the two approaches (Theunissen *et al.*, 2005). Appendix F of this dissertation will describe the underlying values which Popper's open society and FOSSD share in common, such as openness, flexibility and transparency in the face of continuous change. By way of background to that discussion, which may be considered an aside related to the main topic

of this dissertation, the following related work is provided.

Dafermos (2004) suggests there may be a causal link between Popper's philosophy and FOSS. He argues that ever since Popper promoted the notion of an *open society*, "the concept of the value of openness has enjoyed an increasingly growing and widening base of following", which has also impacted the sphere of technology and science. In particular, the author argues, open systems thinking "paved the way for what has come to be known as Free/Open Source software, and it is beyond doubt that open systems' superiority in comparison to closed systems lies in their openness". The author goes on to argue that openness, when coupled with legal instruments such as the GNU General Public License (GNU GPL), protects and promotes an ethical attitude towards technology artefacts and lays the foundation for a more socially-responsible scientific terrain. He also raises the critical questions of whether openness is always beneficial and whether it always promotes ethical behaviour, especially if one considers that open technical knowledge has given rise to the widespread availability of skills and potentially hazardous knowledge at negligible cost. Despite these concerns, the author believes that an important way of ensuring digital ethics is through licensing mechanisms. He promotes two licenses in particular, *The Hacktivismo Enhanced Source Software License Agreement (HESSLA)* and *The Common Good Public License* both of which have human rights at the centre of their concerns. The author concludes that both these licenses seem to have fulfilled their *raison d'etre* and that "I may even make the bold claim that Karl Popper, were he still in life today, would agree".

Asay (2008) believes that OSS closely follows scientific methodology and that it "operates much more like Popper's falsification principle than Baconian induction". The author argues that traditional software development seems to operate by the Baconian method of attempting to search for truth and to achieve certain knowledge whereas open source software development seems to follow the Popperian approach of 'conjectures and refutations' more closely. Furthermore, the author argues that OSS is critical to IT since:

> Just as Popper analogized from science to society, arguing that society required open, 'falsificationist' debate in order to flourish beyond dangerous dogma, so, too, does good software require that it be written outside the auspices of any one company's exclusive control. Good code, like good government, depends on the ability to perceive and correct inefficiencies and imperfections in the fabric of that code (or society), which ability is lessened by centralized control.

The author concludes that the "open development process enabled by open source is vastly superior to a closed system in providing quality software (both in terms of internal code quality and external validation of which code deserves to remain extant)".

Stalder (2003), in his talk entitled *Open Source as a Social Principle*, argues that OSS combines many notions including "peer review" and, most importantly, an openness similar to the open model of social development which Popper contrasted with a closed model of social development in his critique of totalitarianism after World War II. The author claims that "Open Source Software contains powerful arguments how to make these notions, central to a democratic society, relevant in a technology-dominated era".

Finally, Moroz, a representative of one of the Open Society Institutes (OSIs)[29], describes the key elements which he believes OSS and the open society share in common. These include responsiveness, transparency, responsibility, freedom and human rights, social mobility, and the fallibility of knowledge.

The collective account of the related work in this section has shown that several practitioners have found the Popperian concepts of "falsificationism", "three worlds metaphysics" and "open society" to be useful in describing or providing an ideal for aspects of both the software engineering and Knowledge Management disciplines. However, these Popperian concepts seem to have been used in isolation, and no systematic use of Popper's entire philosophy—his *epistemology* and his *ethics*—seems to exist in the software engineering discipline, especially not in relation to understanding Agile software methodologies. The central aim of this dissertation, therefore, is to use ideas from Popper's entire philosophy in an attempt to better understand Agile software development. In particular, Popper's ideas will be used in an attempt to illuminate the values, principles and practices underlying the most prominent Agile methodology, Extreme Programming (XP).

[29]The OSIs were founded by the Hungarian philanthropist, and world renown financial markets speculator, George Soros, with the aim of propagating and reinforcing Popper's ideas of an open society, shaping public policy and promoting democratic governance, human rights, and economic, legal, and social reform worldwide. As a student of Popper's at the London School of Economics (LSE), Soros was highly influenced by Popper's philosophy. However, Soros (2006) differs from Popper on the direct application of the critical method of science to society because social phenomena are, according to Soros, irreducibly reflexive (whereas physical phenomena are not). A recent example of how Soros' OSIs provide support, was in February 2008 when they were called upon by the independent, non-profit global campaigning organisation known as *Avaaz* to administer the funds raised in support of the Burmese people's efforts to peacefully protest against their brutal military leaders. Avaaz, who works to ensure that the views and values of the world's people inform global decision-making, did not believe that it had the expertise to deal with all the activities involved with managing large amounts of money and, therefore, approached the OSI who had also long been supporting the Burmese people. The report back on this campaign can be found in BRB (2008).

This chapter has provided an account of the literature from the software engineering discipline that uses the ideas of various professional philosophers. In relation to Agile software development, this chapter has specifically shown that many practitioners—especially those affiliated with the Cutter Consortium—have used Kuhnian ideas, either explicitly or implicitly, to explain their new "Agile" approach towards software development. In particular, they have used Kuhn's concepts to describe the shift towards Agile methodologies as a revolutionary "paradigm shift" away from traditional software methodologies. This dissertation aims to investigate whether these practitioners' use of Kuhnian ideas is appropriate, and whether they have used Kuhn's ideas consistently with the way they are described by Kuhn (1970). Another central aim of this dissertation is to investigate whether the philosophy of Kuhn's rival, Popper, is any more useful than Kuhn's in understanding aspects of Agile software development, an investigation which, to the best of my knowledge, has not yet been conducted in the software engineering discipline. This chapter concludes the historical background to this dissertation. The rest of this dissertation will contain, in two parts—*epistemology* and *ethics*—a systematic investigation into whether the respective philosophies of Kuhn and Popper can illuminate certain aspects of Agile software development.

Part II

Epistemology

Part II of this dissertation, which comprises chapters 4 and 5, deals mainly with *epistemology*: "The branch of philosophy concerned with the theory of knowledge"[30]. In particular, an analogy will be drawn between Kuhn and Popper's ideas on the development of scientific knowledge, and the Agile and XP leaders' ideas on the development of software. Further to the discussion of epistemology, another branch of philosophy known as *ontology* will be discussed towards the end of this part, in section 5.3. Ontology is "The branch of metaphysical enquiry concerned with the study of existence itself"[31], and is inextricably linked to *epistemology* since the latter studies the theory of knowledge of what *exists* or what "is".

These related branches of philosophy, however, are distinct from the branch of philosophy dealing with "ethics": *epistemology* and *ontology* involve creating theories about the way the world "is", whereas *ethics* involves creating theories about the way the world "ought" to be. In philosophy, this distinction is commonly known as the *is-ought problem* (IOP, 2008), and was first identified by the Scottish philosopher, David Hume, who was introduced in section 2.2. The problematic relation of 'is' to 'ought' has subsequently been named "Hume's Law" and states that "an 'ought' cannot be deduced from an 'is' ".The branch of philosophy dealing with *ethics*, will be the focus of discussion in Part III of this dissertation.

Before using their respective ideas in the following two chapters, Kuhn and Popper's positions on the *is-ought problem* will be described here since these positions may help to clarify the ideas of Kuhn and Popper, which will be used later in this dissertation.

It is important to note that neither Kuhn nor Popper explicitly state in their original works whether they are describing (is) or prescribing (ought). It is my conjecture that the intention of both philosophers is to *describe* rather than *prescribe* how science proceeds, and that one could arguably say that Kuhn's philosophy tends to *describe* reality more accurately than Popper's philosophy does. Perhaps this is because Popper's descriptions of how science proceeds have been shaped by an "ideal" of the objective standard of truth, such that they have implicitly become *prescriptions* of how he believes science ought to proceed.

Eight years after he published the first edition of *The Structure of Scientific Revolutions*, Kuhn explicitly addressed the *is-ought problem* in the postscript to the second edition. Kuhn

[30]Flew (1984) (p. 109)
[31]Flew (1984) (p. 255)

wrote:

> A few readers of my original text have noticed that I repeatedly pass back and forth between the descriptive and the normative modes, a transition particularly marked in occasional passages that open with, 'But that is not what scientists do', and close by claiming that scientists ought not do so. Some critics claim that I am confusing description with prescription, violating the time-honored philosophical theorem: 'Is' cannot imply 'ought'.
>
> That theorem has, in practice, become a tag, and it is no longer everywhere honored. A number of contemporary philosophers have discovered important contexts in which the normative and the descriptive are inextricably mixed. 'Is' and 'ought' are by no means always so separate as they have seemed. But no recourse to the subtleties of contemporary linguistic philosophy is needed to unravel what has seemed confused about this aspect of my position. The preceding pages present a viewpoint or theory about the nature of science, and, like other philosophies of science, the theory has consequences for the way in which scientists should behave if their enterprise is to succeed. Though it need not be right, any more than any other theory, it provides a legitimate basis for reiterated 'oughts' and 'shoulds'. Conversely, one set of reasons for taking the theory seriously is that scientists, whose methods have been developed and selected for their success, do in fact behave as the theory says they should. My descriptive generalizations are evidence for the theory precisely because they can also be derived from it, whereas on other views of the nature of science they constitute anomalous behavior.

It seems, therefore, that Kuhn collapses the is-ought distinction: his "is" is also as he believed it "ought to be". Both philosophers, therefore, ended up having both *prescriptive* elements of how science "ought" to proceed as well as *descriptive* elements of how science actually proceeds. An important difference, however, lies with their respective use of the word "ought". Kuhn's argument seems to be that *if you want to be a successful scientist, then you ought to behave as I say scientists behave.* Therefore, Kuhn's "ought" does not relate to ethics per se, but rather relates to becoming a successful scientist. If you were to ask Kuhn what is good or bad, he may answer that it is good to be a successful scientist

and bad to be a failed scientist. Popper's view of "ought", on the other hand, seems to be in the service of truth, even if it is unattainable, ever-receding. His view is more broadly ethical: it is good honestly to pursue truth and bad to accept or promote untruth or be indifferent towards the pursuit of truth[32]. Popper's view of "ought" in relation to ethics and Kuhn's silence on ethical issues, will be the topic of investigation in relation to Agile software development, in Part III of this dissertation.

The points raised in this introduction concerning the *is-ought problem* and the distinction between *epistemology* and *ethics* should be borne in mind throughout the remainder of this dissertation.

[32]Of course, for Popper, unlike the positivists, falsity is as—if not more—important than truth, since whereas no theory can be decisively verified, all scientific theories can be decisively falsified. But truth is nonetheless the normative goal of science.

Chapter 4

Critique of the Agile Leaders' Use of Kuhnian Concepts

Section 3.2 of the previous chapter showed that Kuhn's concepts of "revolution" and "paradigm shift" have been used, both explicitly and implicitly, by several leading figures of the Agile software community—especially from the Cutter Consortium—in relation to Agile software methodologies. In particular, it was shown that Kuhn's concept of "paradigm shift" has been used to describe the transition from traditional software methodologies to Agile software methodologies, especially towards Kent Beck's *Extreme Programming (XP)*. It was also shown that Kuhn's concept of "revolution" has been used to describe the fundamental changes in software development that the Agile methodologists claim to have brought about. In an attempt to determine whether the use of these Kuhnian concepts by certain members of the Agile community can be justified, this chapter[1] will assess critically the extent to which Kuhn's revolutionary model of change in science can meaningfully be transferred to Agile software development. The implicit assumption made by members of the Agile community that a software methodology can be equated with a scientific paradigm, will specifically be questioned in this chapter.

In order to contextualise the discussion in this chapter and the next, a distinction will be made from the start between two levels of change, namely *large-scale* and *small-scale* change. It is conjectured that Kuhn's concepts of *scientific revolutions* and *paradigm shift*,

[1]Parts of this chapter have been adapted from Northover *et al.* (2007b), for which I was the main author.

which describe large-scale change, may illuminate why certain members of the Agile community have used Kuhnian concepts to describe radical changes in the software engineering discipline as a whole. On the other hand, it is conjectured that Popper's concepts of "evolutionary epistemology" and "falsificationism", may be more suited to explaining small-scale, cumulative changes that occur in the software engineering discipline.[2]

It is of importance to note at this stage, that Kuhn intends his philosophy to be narrowly applicable to scientific endeavours and communities[3]. Therefore, the legitimacy of using his philosophy within the discipline of software engineering, may reasonably be questioned. However, Kuhn does not *discourage* the use of his ideas in disciplines other than science, although he claims to be puzzled by the fact that they have been received far more enthusiastically in the social and biological sciences and in the humanities than in the physical sciences, since the history of the latter, in Kuhn's opinion, differs so strikingly from that of the former.[4] Kuhn would, therefore, most likely not object to his ideas being used within the software engineering discipline but would warn that such use of his ideas would be outside their intended scope. If one considers the widespread influence of Kuhn's ideas in academia, it is perhaps not surprising that Kuhn's ideas can be found in many disciplines other than physical science. According to Steve Fuller, the founder of a discipline called "social epistemology", Kuhn's "*The Structure of Scientific Revolutions*—originally published in 1962—has sold over a million copies, has been translated into over twenty languages, and has remained for over thirty years one of the ten most cited academic works" and "... [it] was the most influential book on the nature of science in the second half of the 20^{th} century—and arguably, the entire 20^{th} century" (Fuller, 2003).

The main reason for investigating Kuhn's ideas in relation to software methodologies in this chapter, is due to the popularity of Kuhn's catch-phrases in the literature of certain leading figures from the Agile software community. These leading figures have both explicitly and implicitly used Kuhnian ideas in relation to Agile software methodologies. As was mentioned in section 3.2, the most notable of all references for this dissertation is that made by the leader of XP, Kent Beck. In the annotated bibliography of both editions of his book, *Extreme Programming Explained: Embrace Change*, Beck explicitly cites Kuhn's seminal work *The Structure of Scientific Revolutions*. The annotation in the first edition reads "XP

[2]The reader is reminded of the definitions of *evolution* and *revolution* in section 2.1 of this dissertation.
[3]Fuller (2003) (p. 21)
[4]Fuller (2003) (p. 21)

comes as a paradigm shift for some folks. Paradigm shifts have predictable effects. Here are some"[5], and in the second edition it reads: "How paradigms become the dominant paradigm. Paradigm shifts have predictable effects"[6]. In addition to these explicit links of Kuhn's work to XP, Beck acknowledged to me, in private correspondence, that Kuhn's book "*was an influential book on my (i.e. his) thinking*" and that he found Kuhn's ideas useful "*Mostly in the adoption process. Kuhn helped me predict how the market would react to XP and helped me communicate across paradigms*". After describing Kuhn's ideas in the following section, it will become clear that Beck's use of Kuhn's ideas, at least in so far as these statements of his suggests, seems to conflict with what Kuhn argued in his *Structure*: Kuhn never mentioned markets and he specifically argued that different paradigms are incommensurable, making communication between scientists within different paradigms very difficult—if not impossible.[7] Nonetheless, what seems clear from Beck's statements is that he uses Kuhn's ideas to describe large-scale change, which he anticipates that the adoption of XP may bring about. However, this interpretation would only correspond with Beck's approach in the first edition of his book, since, in that edition, Beck prescribed that the XP methodology be adopted holistically—that *all* XP values, principles and practices be adopted in order to truly be practising XP. Beck's approach in the second edition, which he explicitly states in the preface (p. xxii), differs from the approach in the first edition since it advocates the incremental adoption of XP. This distinction is an important one for the topic of this dissertation since it may mean that Kuhn's ideas are only relevant to the first version of XP, and not to the revised second version of XP, which superseded the first. Another important point to note from Beck's statements is that he, along with most of the other Agile advocates who have used Kuhnian ideas, does *not* seem to use Kuhn's ideas in relation to the specific values, principles and practices of XP, nor in relation to the small-scale changes that occur during the practice of everyday XP software development.

Using Kuhn's *The Structure of Scientific Revolutions* as a basis, this chapter aims to investigate whether the members of the Agile software community who have used Kuhn's ideas, have done so consistently with the way Kuhn described his methodology in *Structure*. The chapter will take to its logical conclusion, the implications of the systematic application of Kuhn's ideas to Agile software development. Finally, the chapter will investigate the

[5]Beck & Andres (2000) (Annotated bibliography)
[6]Beck & Andres (2005) (Annotated bibliography)
[7]Kuhn (1970) (pp. 44, 94)

reasons for the popularity of Kuhn's concepts within the Agile community, such that the philosophical basis of the Agile software methodologies may possibly be understood better.

4.1 Kuhn's *Structure of Scientific Revolutions*

Kuhn's first book was entitled *The Copernican Revolution* and in it he started developing his ideas that culminated in *The Structure of Scientific Revolutions*. In the latter book, Kuhn explicitly compares scientific and political revolutions: "Like the choice between competing political institutions, that between competing paradigms proves to be a choice between incompatible modes of community life"[8]. For Kuhn, the term "paradigm" means two things[9]:

> On the one hand, it stands for the entire constellation of beliefs, values, techniques, and so on shared by the members of a given [scientific] community. On the other, it denotes one sort of element, in that constellation, the concrete puzzle-solutions which, employed as models or examples[10], can replace explicit rules as a basis for the solution of the remaining puzzles of normal science.[11]

The clearest examples of Kuhn's "paradigm shifts" come from cosmology, namely, the Aristotelian-Ptolemaic, Copernican-Newtonian, and the Einsteinian-Heisenbergian revolutions.

Kuhn divides the development of science into several distinct phases, each leading ineluctably, in an endless cycle, to the next: *normal science*, *crisis*, *extra-ordinary science*, *paradigm shift*, and *new normal science*.[12]

Kuhn argues that a discipline becomes scientific only once *all* the practitioners adopt a single paradigm.[13] The *pre-scientific* period is characterised by several competing schools

[8] Kuhn (1970) (p. 94)

[9] The fact that paradigm means two things is clearly stated in the *postscript* of Kuhn (1970) (the second edition of Kuhn's *Structure*) in response to "one sympathetic reader, who shares my conviction that 'paradigm' names the central philosophical elements of the book, prepared a partial analytic index and concluded that the term is used in at least twenty-two different ways." (Kuhn, 1970) (p. 181).

[10] These exemplar solutions, if interpreted as patterns for solving puzzles in the paradigm, can be compared to software design patterns. This analogy would mean that our collection of software design patterns could be interpreted as the models that define our current paradigms in software engineering.

[11] Kuhn (1970) (p. 175)

[12] This scheme is not a form of Hegelian dialectics, nor a form of Popper's model of evolutionary epistemology, appearances to the contrary, since despite its cyclical nature there is no progress when one paradigm replaces another, and since it is inherently conservative rather than dynamic: its normal state (of normal science) is one of stasis, or *homeostasis*.

[13] Kuhn (1970) (pp. 18-19)

of thought within a single discipline (rather than competing theories, as Popper would have it). Once a paradigm is adopted, all practitioners within it become members of a scientific community, whose research is governed by the paradigm. Kuhn calls this *mature science* and describes the paradigm-governed activities of its scientists as *normal science*, which occurs uninterrupted for long periods of time. New members are induced, or initiated, into the paradigm not primarily through the explicit acquisition of the theories, rules and criteria of the paradigm, but rather implicitly through studying textbook exemplars and practising in laboratories[14], a process of learning that Kuhn calls *tacit knowledge*, explicitly acknowledging Michael Polanyi.[15] This is an efficient way of training new scientists in the paradigm, but it does not encourage open-minded, reflective, self-critical scientists. Kuhn's view of scientists stands in strong contrast to Popper's vision of them as creative, heroic individuals. As Fuller writes, "Scientists are not taught to be mentally flexible"[16]. Also, Kuhn notes that "Mopping-up operations are what engage most scientists throughout their careers" and "Nor do scientists normally aim to invent new theories, and they are often intolerant of those invented by others"[17].

Furthermore, Kuhn contends, referring to George Orwell's *Nineteen Eighty-Four*[18], that the science textbooks are re-written by members of the triumphant paradigm to suggest that all previous scientific work had evolved rationally to culminate in the present paradigm[19]. This contradicts the historical facts. According to Kuhn, "The depreciation of historical fact is deeply, and probably functionally, ingrained in the ideology of the scientific profession"[20]. The reason for this historical revisionism is to indoctrinate the science students and to propagate the myth of progress in science, which emphasises the non-rational factors involved in teaching the paradigm. The fact is, that the old paradigm dies out with the older generation, whereas the new one survives thanks to the indoctrination of the new generation[21].

[14]This process whereby apprentices learn, through imitation, from skilled masters can also be found in other disciplines, for instance, carpentry, hairdressing and bricklaying.

[15]Kuhn (1970) (p. 44). Both Kuhn and Polanyi are said to be the main contemporary protagonists of the belief in "truth by consensus". This is an important point for section 5.3 of this dissertation, which will discuss, in relation to Agile methodologies, the differing viewpoints of Popper and Polanyi regarding truth and knowledge.

[16]Fuller (2003) (p. 38)

[17]Kuhn (1970) (p. 24)

[18]Kuhn (1970) (p. 167)

[19]Kuhn (1970) (pp. 137-140)

[20]Kuhn (1970) (p. 138)

[21]Kuhn (1970) (p. 152)

During *normal science*, scientists do not attempt to refute hypotheses, theories or the paradigm, but rather engage in puzzle-solving (as opposed to Popper's problem-solving)[22]. They resist counter-instances (anomalies) until so many have accumulated that they can no longer be ignored and the scientific community is plunged into a state of *crisis*[23]. Kuhn would consider these facts to be a refutation of Popper's falsificationism, at least as an account of the actual history of scientific discovery. Indeed Kuhn explicitly rejects Popper's principle of falsificationism:

> A very different approach to this whole network of problems has been developed by Karl R. Popper who denies the existence of any verification procedures at all. Instead, he emphasizes the importance of falsification, i.e., of the test that, because its outcome is negative, necessitates the rejection of an established theory. Clearly, the role thus attributed to falsification is much like the one this essay assigns to anomalous experiences, i.e., to experiences that, by invoking crisis, prepare the way for a new theory. Nevertheless, anomalous experiences may not be identified with falsifying ones. Indeed, I doubt that the latter exist.[24]

During a period of crisis scientists practice *extra-ordinary science*.[25] Kuhn argues that no scientific community will abandon the dominant paradigm unless a new one becomes available, since to abandon the paradigm is to abandon science altogether, because all scientific activity occurs within a paradigm[26].

The attainment of a new paradigm is considered *revolutionary* because the break with the old paradigm is final and largely irrational. The very "rules of the game" (a term Kuhn takes from Wittgenstein) have changed[27]. It is a complete change of worldview and is *holistic* rather than *piecemeal*. According to Kuhn:

> Paradigms are not corrigible by normal science at all. Instead, as we have already seen, normal science ultimately leads only to the recognition of anomalies and

[22] Kuhn (1970) (pp. 24, 36)
[23] Kuhn (1970) (p. 77)
[24] (Kuhn, 1970) (p. 146)
[25] It is during this stage only that scientists behave in the way Popper describes, namely, creatively forming numerous competing theories and subjecting them to severe criticism and testing. This is not meant to suggest that Popper's theory is contained in Kuhn's, because they are otherwise radically different, and because Popper's is far more expansive than Kuhn's, both epistemologically and normatively. For instance, all of Kuhn's abandoned paradigms would inhabit Popper's World 3, and become part of the bigger scheme of Popper's evolutionary epistemology.
[26] Kuhn (1970) (p. 79)
[27] Kuhn (1970) (pp. 40, 44-45); Wittgenstein (1958)

> to crises. And these are terminated, not by deliberation and interpretation, but by a relatively sudden and unstructured event like the gestalt switch. Scientists then often speak of the "scales falling from the eyes" or of the "lightning flash" that "inundates" a previously obscure puzzle, enabling its components to be seen in a new way that for the first time permits its solution.[28]

Furthermore, the new paradigm is incommensurable (cannot be compared) with the old paradigm, so much so that scientists within different paradigms are unable to understand each other.[29] Translation between paradigms is very difficult since the scientists belong to different *language communities* (a concept also from Wittgenstein)[30]. "Communication across the revolutionary divide is inevitably partial"[31], according to Kuhn.

Kuhn argues that scientific progress only occurs during normal science within a paradigm, in terms of the cumulative solving of puzzles[32]. Kuhn doubts that science progresses as a whole towards a closer approximation of the truth, although he points out that scientists within the triumphant paradigm would consider the adoption of their paradigm as progress.[33] For Kuhn, persuasion by argument alone is not decisive, but rather certain *non-rational* factors are required to convert people to the new paradigm, particularly faith.[34]

Whereas many people think Kuhn's views of science superseded Popper's, Steve Fuller, in a recent book entitled *Kuhn vs Popper: The Struggle for the Soul of Science*[35], has come to criticise this view: "While no one doubts that Kuhn has won the debate, I intend to question whether it has been for the better" and "With the defeat of Popper (and his followers), the normative structure of science drastically changed"[36]. The ethical implications of Kuhn's supposed victory will be discussed later in chapter III where, firstly, its impact on the democratic accountability of science and scientists will be investigated and, secondly, its impact on the democratic aspirations of those members of the Agile software development community who have used Kuhnian ideas, will be discussed.

---

[28] Kuhn (1970) (p. 122)
[29] Kuhn (1970) (p. 149); Wittgenstein (1958)
[30] Kuhn (1970) (p. 94); Wittgenstein (1958)
[31] Kuhn (1970) (p. 149)
[32] "Puzzle-solving" is another idea from Wittgenstein, who, according to Kuhn (1970) (p. 38) and Quinton (1964) (pp. 542-543), believes that there are no genuine problems in philosophy, merely confusions about language that could be resolved linguistically.
[33] Kuhn (1970) (p. 166)
[34] Kuhn (1970) (pp. 152, 158-159)
[35] Fuller (2003)
[36] Fuller (2003) (pp. 4, 5)

The remainder of this chapter aims to critically analyse the appropriateness of using Kuhn's ideas, as they have been described above, to explain the shift in software methodology from more traditional methodologies to more Agile ones. In particular, Kuhnian terminology will be used to assess whether XP can be described as one of several methodologies which belongs to the *new paradigm* known as "agile", "lean" or "lightweight", where the "classical" or "traditional" software engineering methodologies would be considered part of the *old paradigm*. Consequently, the question will be raised whether Kuhn's theory can adequately explain the shift from classical to Agile software methodologies, especially to XP, and whether this shift can best be described as a "revolution" or "paradigm shift", in the Kuhnian sense.

The following section will provide a critique of the applicability of Kuhn's ideas to software engineering in general, and to Agile methodologies in particular. Thereafter, a detailed and focused critique of the work of Yourdon, a highly influential figure in software engineering, and the most prominent Agile representative in terms of his explicit use of certain of Kuhn's ideas in relation to Agile methodologies, will be provided.

4.2 General Critique of Agile's Use of Kuhnian Concepts

One can ask, as an initial critical question, *whether software developers can be rightfully described as scientific researchers* at all. It seems more obvious that software developers are engineers of sorts, since they produce something that has direct application to the ordinary world, whereas physical scientists are concerned with specialised research which often has no obvious bearing on everyday life. Whereas the engineer synthesizes/constructs to produce new artefacts, the scientist analyzes/takes apart to acquire knowledge about an existing (natural) entity. Furthermore, scientists are not normally held accountable to the public to the same extent as software engineers, who do not merely solve theoretical problems but produce software with clear applications. Nonetheless, both make use of the rigorous standards of modern *mathematical logic*, and computer technology has become essential to modern scientific research as well.[37]

[37] A similar distinction between science and technology/engineering is made by Arageorgis & Baltas (1989). Those authors argue that "science aims at increasing and rationalizing knowledge by establishing better and better theories. The purpose of scientific research is increased understanding. On the other hand, technology aims at satisfying human and social needs (i.e. realizing desired states of affairs) through devising material artifacts which are appropriate and as effective as possible. The purpose of technological research is usefulness" (p. 212).

It can also be questioned whether *software methodologies can rightfully be regarded as paradigms*, as Beck, for example does when he describes "This is the paradigm of XP"[38]. Software practice can be made to fit, with minor modifications, Kuhn's second meaning of the word paradigm as defined earlier in this chapter: "concrete puzzle-solutions . . . employed as models or examples. . . ". It also appears to fit his first definition of a paradigm as "the entire constellation of beliefs, values, techniques, and so on shared by the members of a given community". Therefore, software practice seems to fit both of Kuhn's definitions of a paradigm, albeit in a broad sense. However, what is not evident from Kuhn's definition is that he was writing specifically about *scientific* communities, and, even more specifically, communities of *physical scientists*. Taken in this specific and historic sense, it is doubtful that the term can be applied to software communities without modifying its meaning.

For Kuhn, crisis in the paradigm can be precipitated in three possible ways, namely, with the accumulation of anomalies, with a new discovery, or with a new theory, in this order of likelihood. Therefore, one could *question which of these caused the software crisis*[39]. It seems as though none of them are literally applicable. Instead, the software crisis began in the 1960s (as described in detail in section 2.3) when, for the first time in computer history, software became more costly than hardware. The frequency of over-budget and over-schedule projects, some of which also caused damage to property and even resulted in the loss of life, increased dramatically. The crisis was arguably further worsened by the dawn of the Internet era, which saw traditional software methodologies struggle to cope with the demand for rapid and continuous change, and the consequent urgent need for new software methodologies that would be able to cope with the challenges of the Internet era. The long-standing software crisis and the attempt by the proponents of the so-called "lightweight" software methodologies to solve this crisis, has been interpreted by certain members of the Agile software community in terms of Kuhn's "scientific crisis" or "extra-ordinary science". These members see Agile methodologies as the emerging "paradigm", which will completely replace the old "paradigm" of traditional methodologies.

If it is appropriate to compare scientific paradigms to software development methodologies, then the questioning of the fundamental values, principles and practices of a software methodology that accompanies the emergence of a new methodology can be compared to

[38] Beck & Andres (2005) (p. 11)
[39] *SCR (2008)*

the similar activity of physical scientists in a state of crisis. However, this presupposes that the software engineering discipline is already a "mature science", that a single paradigm does, in fact, dominate, and that the discipline is not in a "pre-scientific" state, characterised by many competing schools of thought. If one considers that the software crisis has existed since the mid 1960s and that prior to the 1960s there was no recognised software methodology, it seems fair to say that no single software methodology or "paradigm" has ever dominated the software engineering discipline[40]. Therefore, the discipline cannot be regarded a "mature science" in the strict Kuhnian sense of the term, nor does Kuhn's term "scientific crisis" seem applicable to software engineering because a crisis period, according to Kuhn, is presupposed by a dominant paradigm and a "mature science". Instead, to use Kuhnian terminology, the software engineering discipline seems still to be in a "pre-scientific" state since there are several competing software methodologies in use today as opposed to a single, dominant one[41]. Therefore, Kuhn's notions of "paradigm shift" and "crisis" cannot be used in their strict sense to describe the software crisis and the emergence of the lightweight methodologies as the "silver bullet" (Brooks, 1987) to solve this crisis. Furthermore, the assumption that there need be a single dominant paradigm in software engineering as Kuhn argues there is in science, can itself be questioned. As discussed in section 2.3, Cockburn's suggestion of "a methodology per project" (PPP, 1999) is one which rejects an approach of a single methodology for all projects. Cockburn's approach is thoroughly non-Kuhnian since it implies that rational choices can be made between different software methodologies for each new project depending on the project's requirements.

Furthermore, *are software methodologies really incommensurable and an all-or-nothing affair, as Kuhn alleges scientific paradigms to be*? The existence of numerous articles in the software literature that compare different software methodologies, suggests that they are not incommensurable. Boehm (2002), for example, provides a detailed comparison between Agile methodologies and plan-driven software methodologies and argues for their synthesis into a hybrid. Similarly, PLW (2003) claims to have successfully adapted the *Waterfall*

[40] A case can be made for the *Waterfall Model* being dominant in the 1970s and 1980s, however, since this model was more suited to large military projects, it is doubtful that *all* software projects, including smaller ones, used the Waterfall model during this period.

[41] Even within a single organisation, it is not uncommon to find different software methodologies co-existing. For instance, the old mainframe way of developing software may co-exist with the new distributed way, as well as with offshore development, and vendor package integration. Each of these has their own way of thinking, and their own "language". Therefore, it is usually one of the major challenges in a typical technology-spanning project to get the different methodologists to work together as a single team.

Model by integrating key elements of approaches like "Rapid Application Development" (RAD) and XP, which are traditionally considered to be diametrically opposed to the *Waterfall Model*. Furthermore, Paulk (2001) discusses XP practices in relation to the CMM key process areas (KPAs) and argues that, taken together, the two methods can create synergy. Finally, Jeffries (2000) shows that XP has some characteristics in common with the five CMM levels[42].

Another important question in this context is: *"Was the emergence of Agile methodologies in the mid 1990s a result of cumulative anomalies?"* For Kuhn, anomalies could take the form of "discoveries, or novelties of fact"[43] on the one hand, or "inventions, or novelties of theories"[44], on the other. Both forms of novelty, however, are usually *not* actively sought out by—and, in fact, are initially fiercely resisted by—scientific communities. Since, software engineering methodologies do not aim to explain physical phenomena, it is difficult to see how Kuhn's theory is applicable to them in this case.[45] Nonetheless, a case can be made for crucial events exacerbating the number of anomalies (failed software projects) in software engineering. The advent of the Internet era, which saw an increase in the number of failed projects due to their inability to adapt to continuous change, may be one such event. At that time, object-oriented (OO) software methodologies were the dominant methodology (or "paradigm", to use Kuhnian terminology). Most software engineers came to recognise that developing at "Internet time" required a reconsideration of the object-oriented software process. In order to create truly new classes of software, which would better deal with the challenges[46] of the Internet era, the classical software process needed to be radically overhauled. As a result, the lightweight or Agile methodologies emerged and many of the advocates from the old object-oriented "paradigm" became advocates of the new Agile "paradigm".

The emergence of the software engineering discipline itself—as a distinct discipline from

[42]It is likely that XP shares more in common with the Testing Maturity Model (TMM, 2002)—which was created to complement the CMM as discussed in section 2.3—since both are focused fundamentally on rigorous testing.

[43]Kuhn (1970) (p. 52)

[44]Kuhn (1970) (p. 52)

[45]Arageorgis & Baltas (1989), while maintaining that science and engineering are independent disciplines, note that the two disciplines are intimately related since, firstly, "A need may be raised in the interior of a given science appealing to the realization of a particular artifact", and secondly, "A need may be raised outside a give science in various quarters and for various reasons and appeal to that science for its satisfaction" (p. 219).

[46]The challenges included *rapid technological change*, *volatile business requirements*, *increasing scale and complexity* and *shortening market time windows*.

computer science about forty years ago—can, in a similar way to the emergence of Agile methodologies, be considered the result of a crisis. Ever since, software engineers have been living with this software crisis, which suggests that the perpetuation of Kuhnian "anomalies" have become a new kind of normality in the software engineering discipline.

Another relevant question in this discourse is: *"Can Agile methodologies justifiably be termed revolutionary if they have not yet become the dominant software paradigm in the ten years since their inception?"*[47] Kuhn himself faced a similar criticism: "The period between Copernicus and Newton is often termed 'The Scientific Revolution', but the time-span involved, over 150 years, makes the process sound more like evolution than revolution" (Bullock & Trombley, 1999). On the other hand, the definition of "revolution" from section 2.1 does not refer to time-span at all. Whereas political revolutions seem to occur swiftly, scientific and technological revolutions can take considerable time to occur, as was evident in, for instance, the "Industrial Revolution". Nonetheless, given the swift rate of change in the Information Age, perhaps the Agile methodology "revolution" should have occurred already, if it is to take place at all. The most important question in this context, remains open: "Is software engineering actually a science already, or is it still in a state of pre-scientific (although technical) human practice?"[48]

Furthermore, *"Is the practice of Agile software development similar to Kuhnian normal science?"* Wernick & Hall (2004) suggest that it may be similar insofar as the uncritical adherence to any method or tool parallels Kuhnian *normal science*:

> This is a reasonable parallel to what is required of the software developer who uses a pre-defined method or tool. He or she should concentrate on using that mechanism to develop his or her product, rather than seeking to explore the weaknesses of the mechanism. Therefore, any instance of software engineering practice based on pre-defined mechanisms can reasonably be seen as being similar to Kuhnian 'normal science'. (p. 5)

[47] Knuth's seminal book *The Art of Computer Programming* (1st ed. 1968) prompted Gries' book *The Science of Programming* (1st ed. 1981), which indicates how slowly the software engineering community is trying to come to terms with an appropriate philosophical understanding of the practices and activities in their own discipline.

[48] Arageorgis & Baltas (1989) argue for a strict demarcation between science and engineering. They believe every "technological problem lies outside the jurisdiction of any single science" because "each science can treat only the one particular aspect of a real phenomenon which is idiosyncratic to it" (p. 219). Therefore, these authors would consider it incorrect to call software engineering a science. Software engineers do, however, use the various sciences as a means to the ends of producing material artifacts.

The Agile community will most likely object to this interpretation since, rather than viewing themselves as uncritical adherents to Agile methodologies, they position themselves as open to criticism and continuous change. With regard to XP in particular, a case can be made for interpreting the fact that many software engineers have adopted XP holistically—based on the advice of Beck in the *first* edition of his *Extreme Programming Explained* (Beck & Andres, 2000)—as analogous with the way in which scientists operate during *normal science*, namely, by adopting the practices of their paradigm wholeheartedly and unquestioningly. Note, however, that this point relates specifically to the way in which software engineers have *adopted* XP based on the advice in the first edition of Beck's book. It does not relate to either the advice in the second edition of Beck's book, nor to the values underlying the XP methodology's principles and practices themselves, which, as will be shown in the following chapter, are opposed to an uncritical and inflexible approach.

An important difference regarding the adoption of XP versus the adoption of a new scientific paradigm, is worth noting here. Kuhn argues that before scientists will be convinced to adopt a new paradigm, there *must* exist at least one successful paradigm case. Often a single strikingly successful paradigm case is all that is needed to make the new paradigm very convincing. In the case of XP, however, the C3 project[49] during which Beck formulated the XP methodology, instead of being the striking "paradigm case" for the XP paradigm, was eventually deemed a failure in 2000 and canceled completely. Despite the apparent failure of the C3 project, it is cited by many, Beck himself included, as the project that gave birth to XP[50]. This account of XP's origins seems to be at odds with Kuhn's belief in a successful paradigm case, however, Kuhn did also say that a paradigm struggles for some time to become dominant, until a successful paradigm case arises[51]. It can be argued that this is what happened in the case of XP—the C3 project may just have been one of the necessary failed paradigm cases which nonetheless contributed towards bringing about the XP 'paradigm shift'.

Similarly to Kuhnian scientists, Agile methodologists rely heavily on *tacit* knowledge, and a case can be made for interpreting the way in which the practice of Agile software development is learnt, as being similar to the way in which scientists learn a paradigm using textbook exemplars. Indeed, Beck states: "The values, principles and practices of XP are

[49] CCC (2008)
[50] Beck & Andres (2005) (p. 125)
[51] Kuhn (1970) (p. 169)

best learned by example".

Finally, one can ask *how far one can speak of paradigm shift in relation to software methodologies themselves rather than their final products?* In physics, it seems that the methodology itself does not change from one paradigm to the next but rather the final product changes[52]. Nonetheless, it is in fact not so easy to separate the methodology from the theory. For instance, in cosmology, the shift from a Newtonian to an Einsteinian paradigm did not merely lead to fundamentally different ways of perceiving physical change but was also accompanied by radically different ways of testing within the different paradigms, including significant changes in technology and tools. Similarly in software engineering, improved tools such as optimised compilers radically influenced software methodologies which, in turn, produced different and more complex final software products.

After careful consideration in this section, it seems that, in their broadest possible sense, certain of Kuhn's ideas seem to be useful in explaining aspects of Agile software methodologies (especially with regard to understanding the adoption and propagation of these methodologies). However, as soon as Kuhn's ideas are used in their specific sense (i.e. in the way Kuhn intended them to be used), they seem to be less relevant to Agile software methodologies. It is my conjecture that many of the Agile leaders who have used Kuhnian ideas, have done so at a broad level, without having a deep knowledge or understanding of Kuhn's philosophy. These leaders seem to have conveniently used Kuhn's popular buzzwords to further their own agenda, namely, to propagate their methodologies, without realising that many of Kuhn's ideas conflict with the stated core values and principles of their methodologies.

4.3 Specific Critique of Yourdon's Use of Kuhn's Concepts

In section 3.2 of the related work chapter, it was shown that several authors from the software engineering community have either explicitly used Kuhnian concepts in their discipline or have used Kuhnian terminology inadvertently to describe significant events in their discipline. In particular, the review focused on the literature from several members of the Cutter Consortium and it was said that Yourdon's writings would be analysed more deeply since

[52]The new paradigm may still use the same methodology as the old paradigm—a deductive scientific method, for example. However, the new paradigm may have brought about new ways of working within the paradigm, which means the final product (i.e. the scientific theory) will differ from the final product of the old paradigm.

he is Cutter's most prominent and influential member as far as applying Kuhn's work to Agile software methodologies is concerned. Besides this reason for specifically investigating Yourdon's writings, there are several additional reasons to justify such an investigation. Firstly, as a co-founder and senior consultant of the Cutter Consortium, Yourdon is highly influential within the Agile community. He has written prolifically, having "published 27 computer-related books and over 550 technical articles" (EY1, 2008). Secondly, according to DMA (1997), "Yourdon invented much of modern software engineering in the 1970s. He has produced a steady stream of books for the past 20-plus years, many of them reporting on the latest trends in the field". Finally, according to the Cutter Consortium, "Yourdon is widely known as the lead developer of the structured analysis/design methods of the 1970s" and "During his career, he has been involved in a number of pioneering computer technologies, such as time-sharing operating systems and virtual memory systems" (EY2, 2008). In 1997, he was inducted into the *Computer Hall of Fame*. The aim of this section is to provide an in-depth analysis of Yourdon's writings which relate to Kuhn, in an attempt to determine the extent of Kuhn's influence on the Agile methodologies through Yourdon's writings.

Before proceeding with the investigation of Yourdon's use of Kuhn's ideas, the question can be raised why certain members of the Agile and XP software communities find Kuhn's ideas so attractive, especially when it is clear that their use of Kuhn's ideas is selective, and their understanding of his ideas is often inaccurate. Firstly, the reader is reminded that Kuhn's *Structure* was hugely influential in most academic disciplines, especially in the United States but also worldwide, on academics who were otherwise unaware of any philosophical alternatives. Secondly, Kuhn's ideas seem to suit these Agile proponents' strategic objectives, namely to initiate a revolution or paradigm shift in software methodology and to change the way software engineering is practised. Thirdly, his ideas apparently support their self-perception of being revolutionaries with a historical purpose. Their interpretation of Kuhn as a revolutionary spirit is thoroughly mistaken though, but it nonetheless may form part of the self-image of certain members of the Agile community.

In spite of his various references to Kuhn, Yourdon's account of Kuhn's ideas is often inaccurate and seems to be influenced by the same revolutionary (or, rather, rebellious) spirit identified in Schwaber's article in section 3.2, a spirit which runs contrary to that of Kuhn's *Structure of Scientific Revolutions*, its title notwithstanding.

Yourdon provides a definition of "paradigm" that departs significantly from Kuhn's first definition of the term, when he writes of paradigms that they "perform a reasonably good job of describing and explaining the events and phenomena that we encounter in our day-to-day life" (Yourdon, 2001d). This misses Kuhn's crucial point concerning the "unparalleled insulation of mature scientific communities from the demands of the laity and of everyday life"[53] that a paradigm refers to a highly specialised scientific community, dealing not with everyday phenomena, but with unfamiliar, even obscure, aspects of nature that few people outside of that scientific community can comprehend. Most software engineers are not as isolated from everyday experience as most scientists. Instead, they, unlike scientists, are held accountable to the public since they produce artefacts for use in everyday life. Furthermore, as in XP's case, stakeholders of a software project—including customers—interact daily with software engineers, which indicates that software teams do not form isolated communities.

Yourdon's definition of a paradigm also omits Kuhn's second sense of the term, namely, "the concrete puzzle-solutions which, employed as models or examples, can replace explicit rules as a basis for the solution of the remaining puzzles of normal science"[54]. This omission is surprising, given that, implicit in the definition, is the concept of *tacit knowledge*, an important concept to Agile software development. Like student-scientists, new software engineers will be initiated into the paradigm (in Kuhn's first sense) more by studying previous puzzle-solutions (successful pieces of software) which they will be able to apply in different puzzle-solving scenarios, than by explicitly learning the principles and rules governing it.[55] It is doubtful, however, that software textbooks and courses indoctrinate students to the same extent that Kuhn claims science textbooks do[56].

According to Kuhn, during normal science, a single dominant paradigm prevails. This dominant paradigm governs the activities within the paradigm—the methodology—which means that there is only ever a single methodology during Kuhnian normal science. If a paradigm is equated with a software methodology (as Yourdon does by stating "Beck's XP approach—which is merely one of several examples of a new paradigm that is coming to

[53] Kuhn (1970) (p. 164)
[54] Kuhn (1970) (p. 175)
[55] While the values, principles and practices of XP are all explicit, Kuhn would say these are usually acquired tacitly, at least during normal science. During times of crisis and extraordinary science, however, the values, principles and practices would be spoken about explicitly, so Kuhn's ideas would not be entirely inappropriate here.
[56] Kuhn (1970) (p. 138)

be known as an 'agile', 'lean' or 'light' methodology. . . "), it seems fair to say that software students, unlike science students, are exposed to alternative methodologies, analogous to the competing schools of Kuhn's pre-scientific phase.

Moreover, Yourdon's reference to "scientists, engineers, soothsayers, or priests" (Yourdon, 2001d) is also very problematic, since Kuhn's "paradigm shift" specifically relates only to physical scientists. At no point in *Structure* does Kuhn refer to engineers, and certainly not to soothsayers and priests, since he considered himself narrowly to be a historian of the physical sciences. Nonetheless, the reference to soothsayers and priests, if not true to the word of Kuhn's theory, is close to its spirit, since it suggests that changes between paradigms are not completely rational, but require a leap of faith.

Similarly, one could object to Yourdon's descriptions that "Meanwhile, there's likely to be a band of renegade scientists, engineers, or priests looking for a new paradigm" and "In the past, the rebels promoting a new paradigm were likely to be burnt at the stake" (Yourdon, 2001d). Yourdon's use of the term "revolution" echoes Schwaber's use of the same term in section 3.2.

Fuller, to the contrary, points out that Kuhn's enthusiastic followers "ignored that Kuhn, far from being a 'scientific revolutionary', argued that revolutions were only a last resort in science—indeed, an indication of just how fixated scientists tend to be on their paradigm [is] that they have no regular procedure for considering fundamental changes in research direction"[57]. Yourdon seems to have shifted from Kuhn's scientific sense of "revolution" to a political sense of the word which is, in fact, alien to Kuhn's theory (despite the fact that Kuhn actually compared the two). From Fuller's critique, it is evident how close Kuhn's paradigms are to the blueprints[58] advocated by traditional methodologists. This is in complete contrast to XP with its focus on flexibility and its many practices for accommodating change. Furthermore, Yourdon's claim that the new paradigm "explains all of the known phenomena much more cleanly and simply [than the old one]" also misrepresents Kuhn, who argues that the new paradigm does not solve all of the old puzzles, but rather abandons many of them, and, in fact, reviews, in the terms of the new paradigm, puzzles previously thought solved. Indeed, the very perception of the phenomena themselves has changed in

[57] Fuller (2003) (p. 22)

[58] Software blueprints are strict plans (often manifested in a priori designs of architecture and inflexible project plans), which are created at the start of a software project and which detail the steps to be performed in implementing a software solution. The steps are to be carried out in order and no changes to the plan are usually permitted.

the new paradigm, so radical is the break with the old paradigm. On the other hand, the new paradigm solves many of the new puzzles that the older paradigm failed to solve. A final problem with Yourdon's depiction, from a Kuhnian point of view, is that it presents an objective criterion that allows one to make a rational choice between paradigms, a possibility which Kuhn rejected, and which more accurately fits the philosophy of his rival, Popper, as will be described in the following chapter.

The critique in this chapter of the applicability of Kuhn's ideas to Agile software methodologies inevitably raises the question *"Why did several members from the Agile community, most notably Kent Beck, use Kuhn's theory of scientific revolutions?"*

Perhaps these members have used Kuhn's ideas for *predictive* rather than descriptive purposes. Their use of Kuhn's ideas may express a hope (faith) that there will be a paradigm shift, a shift which they, seeing themselves perhaps as revolutionaries and their methodology as a radical departure from the traditional methods of software development, hope to bring about. Perhaps they view the Agile and XP movements as participating in a momentous turning point in history, at least in terms of software engineering.[59]. While this may be their intended use of Kuhnian ideas, it is important to note that, according to Fuller (2003), there is a widely held misconception that Kuhn is a *radical revolutionary*. On the contrary, Kuhn's critics saw him as "the official philosopher of the emerging military-industrial complex"[60] and rather than having killed positivism, "[a]s the Popperians saw it, Kuhn simply replaced the positivist search for timelessly true propositions with historically entrenched practices. Both were inherently uncritical and conformist"[61]. Also according to Fuller (2003), part of the popularity of Kuhn is "the innocence [of his admirers] of any alternative accounts of the history of science... with which to compare Kuhn's"[62]. Therefore, Kuhn was not the revolutionary that the Agile and XP methodologists may have thought he was.

If we consider again Kuhn's insistence that paradigms are incommensurable, the reason for Beck's specific acknowledgment of the influence of Kuhn's ideas on the XP methodology (both in the annotated bibliography of the first and second editions of his *Extreme Programming Explained*, and in private correspondence with me), may become apparent. Kuhn's view would have been that the XP methodology cannot be compared to any other software

[59] If this were the case, the use of the word 'revolution' may be a reference to the American Revolution of 1776—1783.
[60] Fuller (2003) (p. 32)
[61] Fuller (2003) (p. 35)
[62] Fuller (2003) (p. 22)

methodology, which implies that choosing between software methodologies is not completely *rational*. Whereas Beck would most probably argue that all of the XP values, principles and practices themselves are rational, he may well have gained an insight about the *irrationality of paradigm shifts* from Kuhn, and this may have helped him to understand that the rationality of the XP values, principles and practices alone, would not trigger a paradigm shift in software methodology towards an adoption of XP. Instead, certain irrational factors may influence the decision of the software community to adopt XP, for instance, the influence of marketing and promotional material. From Beck's statement at the start of this chapter that "Kuhn helped me predict how the market would react to XP", it can be conjectured that the adoption of XP will **not** be largely due to rationality and good arguments regarding the soundness of the XP values, principles and practices, but will instead require an emotional conversion or *leap of faith*. It seems that Beck may have gained this insight from Kuhn despite the fact that Kuhn never mentioned markets. This may, therefore, be a plausible way in which to understand Beck's use of Kuhnian ideas in the first edition of his book: although XP's values, principles and practices are highly rational, the adoption of the XP methodology as a whole, will be a matter of emotion rather than logic, as full of hype and persuasive techniques as the selling of a new product on the market. However, this does not explain Beck's citation of Kuhn's *Structure* in the second edition of his book. For if the XP methodology can be adopted incrementally—as Beck advocates in the second edition of his book—one can argue that no paradigm shift in methodology need occur, since the individual XP values, principles and practices will be adopted incrementally as needed, without necessitating a complete switch from the practices of the old methodology to those of the XP methodology. The following two chapters will investigate whether Beck's advocacy of the incremental adoption of XP in the second edition of his book, can be explained better by Popper's notions of *evolutionary epistemology* and *piecemeal social engineering*.

In section 6.2 of chapter III, the possible ethical implications for the XP methodology of Kuhn's ideas will be investigated. It will also be investigated whether the use of Kuhnian concepts by certain members of the XP community may not have had detrimental consequences for the ethos of the XP methodology as a whole. The critique will focus on the way in which XP is *propagated* and *adopted*, and will be based on a particular practitioner, Johnson's, critique of XP in this regard.

Chapter 5

Understanding Agile Methodologies using Popper's Philosophy

The previous chapter investigated whether the epistemology of Thomas Kuhn could provide an account, analogous to paradigm shifts within scientific disciplines, of large-scale shifts in software methodologies. Whatever the success of that analogy, Kuhn's ideas seem largely inadequate for explaining the small-scale change that occurs during the day-to-day practice of software development. Therefore, this chapter[1] will use the philosophy of Karl Popper, another influential philosopher of science in the 20^{th} century who dealt with epistemology and the question of *change in science*, to see how far his philosophy can account for small-scale change, and especially the "culture of change" in Agile software methodologies. The aim of the discussion in this chapter is to illuminate the philosophical basis of Agile methodologies by investigating which aspects, if any, of the Agile methodological approach are aligned with Popper's ideas. Any aspects of the Agile approach which are found to contradict a Popperian approach, will also be addressed. In order to provide concrete examples for the arguments in this chapter, an analogy will be drawn between Popper's *epistemology* and the "philosophy" underlying the values, principles and practices of Extreme Programming (XP)—the most prominent Agile methodology. In other words, Popper's ideas will be used in an attempt to illuminate small-scale change during XP software development.

[1]Parts of this chapter have been adapted from Northover *et al.* (2006), for which I was the main author.

At the end of the previous chapter it was said that the critique provided there regarding the applicability of Kuhn's ideas to Agile software methodologies would be extended in section 6.2. It was also said that this extended critique would be independently corroborated by the critique of XP that Johnson (2006) makes throughout his book, *Hacknot*. Regarding this corroboration, it is important to note that Johnson's critique of XP seems to focus on the way XP is *propagated*, and does not include the stated values, principles and practices underlying the XP software methodology itself. In particular, Johnson objects to the unfalsifiable truth claims that many XP leaders make when encouraging the *adoption* of XP.[2] This chapter will investigate whether the unfalsifiability of these truth claims may, at least partly, be attributed to the (explicit or implicit) influence of Kuhn's ideas on certain XP leaders. Using Popper's ideas, this chapter will also investigate the underlying "philosophy" of the XP values, principles and practices, which may also illuminate the reason Johnson criticises the *propagation* and *adoption* of XP but not the values, principles and practices of the methodology. At the end of this chapter, it will be shown that Johnson, without acknowledging Popper, uses several Popperian ideas to criticise the way XP is propagated. The novel contribution made by that part of the chapter involves extending Johnson's critique and clarifying his use of Popperian ideas.

This chapter contains three main sections which correspond to three of the central aspects of Popper's philosophy: *The Philosophy of Science*, *Evolutionary Theory of Knowledge* and *Metaphysics*[3]. The section on the *The Philosophy of Science* will use Popper's principle of *falsificationism*, firstly, to investigate whether the software engineering discipline resembles a scientific discipline, due to the susceptibility of software to testing, and, secondly, to investigate the extent to which the XP values, principles and practices are aligned with Popper's principle of falsificationism, and his related ideas of *criticism* and *error elimination*. In an attempt to illuminate the similarities between a software methodology, and a scientific methodology, the section on the *Evolutionary Theory of Knowledge* will investigate the similarities between Popper's *four-stage model* of the way scientific knowledge advances, and the *iterative and incremental* software development approach advocated by the Agile and XP software methodologists. The third, and final, section of this chapter will

[2] An example of an unfalsifiable truth claim, according to (Williams *et al.*, 2000), is "Pair programming results in better code".

[3] Another central aspect of Popper's philosophy, *The Open Society*, will form the basis for the discussion in Part III of this dissertation, which deals with the social and ethical aspects of Agile software development (as opposed to the epistemological aspects of this chapter).

investigate whether Popper's *three worlds metaphysical model*—which positions *objective* knowledge as the most important form of knowledge—may provide an understanding of how Agile methodologists create and share knowledge, despite their stated reliance on *tacit* or *subjective* knowledge.

Before attempting to investigate these three main aspects of Popper's philosophy in relation to Agile software methodologies, a summary of Popper's epistemology—which was briefly introduced in chapter 2—will first be provided here.

Popper calls his philosophy "critical rationalism" and associates it with the tradition of fallibilism since it rejects the quest for secure foundations (justifications) for knowledge. Although mistakenly perceived to be a neo-positivist, Popper is one of the strongest critics of neo-positivism's attempts[4] to justify scientific discoveries using the principle of verificationism. Popper points out that, while no number of confirmations is sufficient to prove (or verify) a scientific generalisation (or theory), only one counter-instance is logically required to disprove (falsify or refute) the theory. Thus, for Popper, falsificationism is the critical method of scientific discovery, and good scientific practice consists in rigorous attempts to falsify, test and criticise creatively formed hypotheses. Popper contends that the susceptibility of theories to falsification i.e., their potential for being refuted, is what demarcates science from metaphysics. Despite rigorous testing, the truth of a hypothesis or theory, according to Popper, always remains tentative (or conjectural). The deductive scientific method of creative hypothesis formation and rigorous error elimination and falsificationism is what Popper terms his *evolutionary theory of knowledge*[5]—a theory of change in science based on an evolutionary model, in contrast to Kuhn's revolutionary model described in the previous chapter[6].

This chapter will investigate whether Popper's theory of change in science can help provide and understanding of the approach towards change advocated by Agile software methodologists. The reader is reminded that the defining characteristic of all Agile software methodologies is their anticipation of the need for flexibility in the face of change and their

[4]Note that neo-positivism is being referred to here, rather than positivism in its classical form. The reader is referred to section 2.2 where the former variant is discussed in detail.

[5]Popper's evolutionary theory of knowledge is, in fact, much broader than scientific method. However, this dissertation will only consider the theory in relation to scientific method.

[6]Section 5.1 will, however, show that Popper encourages revolutionary activity during certain phases of his evolutionary model, whereas revolutionary activity only occurs in Kuhn's model during the stage of *crisis*. During all other phases of Kuhn's model, especially *normal science*, revolutionary activity is not anticipated.

willingness to *embrace change*. The following excerpts from the literature of the Agile and XP communities highlight their commitment towards change.

The practices of the Agile methodologies support an evolving list of requirements and features, and can adapt to changing technologies while software is being developed in an iterative and incremental fashion. As pointed out in the introduction to this dissertation, one of the four central values[7] of the *Manifesto for Agile Software Development* is "*Responding to change* over following a plan" (Beck *et al.*, 2001a) and one of the supporting principles of the Manifesto is "Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage" (Beck *et al.*, 2001b). Agile methodologists argue that comprehensive plans, in contrast to evolving requirements, lay out in advance all steps that will be performed, which suggests that events are necessarily predictable. However, such predictability, they argue, is rare in software projects. Consequently, the Agile methodologys' practices have been designed to accommodate changing requirements and technologies.

Regarding XP in particular (which will be the Agile methodology of specific focus in this chapter), Beck & Andres (2005) emphasise change at two levels, namely, *technological* and *social*: "XP is my attempt to reconcile humanity and productivity"[8]. Beck places particular emphasis on the social impact of XP by stating "Extreme Programming (XP) is about social change."[9] and "Tools and techniques change often, but they don't change a lot. People, however, change slowly but deeply. The challenge of XP is to encourage deep change".

The specific focus of this chapter, however, will be on the technological aspects of change which are embedded in XP's *values, principles and practices*. In particular, an analogy will be drawn between the technological aspects of change in software engineering and the epistemological aspects of change in science. The social (and ethical) aspects of change will be discussed later, in Part III of this dissertation. In preparation for the discussion in the rest of this chapter, which will refer frequently to the values, principles and practices of XP, the following description of them is paraphrased from Beck & Andres (2005): XP

[7]The reader is reminded that even though the creators of the Manifesto used the term "values", strictly speaking, these "values" would be more appropriately termed *maxims* since they do not relate to ethics, as such. The reader should bear this point in mind when the Agile methodologists' terminology of "values" and "principles", and the XP methodologists' terminology of "values" "principles" and "practices" is used in this dissertation.

[8]Beck & Andres (2005) (p. 3)

[9]Beck & Andres (2005) (p. 1)

practices are the daily tasks performed by XP developers. The *values* of XP, which bring purpose to practices, are those universal maxims which we all have an intuitive sense for. The *principles* of XP are domain-specific guidelines which bridge the gap between the values and practices. A concrete example may help to ground this definition: the XP practice of *pair programming* is underpinned by the XP value of *communication*, which, in turn, is informed by the XP principle of *humanity*. Appendix B contains a complete listing of the XP values, principles and practices[10].

[10]While a philosopher may perhaps not find their characterisation of values, principles and practices satisfactory, this dissertation will use these terms as Beck & Andres (2005) define them.

5.1 The Philosophy of Science

In this section, Popper's principle of *falsificationism* will be transferred from the domain of physical science to the domain of software engineering, by applying it to software testing in general and to XP software development in particular. Whether or not it is legitimate to do so is another philosophical question, one which, as the previous chapter pointed out, has already been raised by authors such as Arageorgis & Baltas (1989). However, the conjecture in this section is that several of the XP values, principles and practices have a strong affinity with Popper's critical method, exemplified by his principle of falsificationism. Before investigating this conjecture, however, an elaboration on Popper's principle of falsificationism, which was first introduced in section 2.2, will be provided below.

5.1.1 Popper's Principle of Falsificationism

Popper rejects the traditional view of scientific method, namely, the inductive method of *verificationism*, in favour of the deductive scientific method of *falsificationism*[11]. Popper also rejects the inductivists' quest for firm foundations in science believing, instead, that scientific theories are always tentative approximations to the truth.[12] The following is a brief summary of the two contrasting scientific methods:

> **Inductivists**, on the one hand, believe that knowledge develops as a result of observation and experiment. They base general statements on accumulated observations of specific instances and search for evidence to confirm or *verify* their theories.
>
> **Falsificationists**, on the other hand, argue that general statements, like the laws of physics, can never be conclusively verified but can be conclusively *falsified* by a single counter-instance. A theory is said to be "falsifiable" if its class of *potential falsifiers*—all those basic statements with which it is inconsistent—is not empty.

[11] The principle of falsificationism, according to Popper (1959), can be regarded as the application of the mathematical technique of *reductio ad absurdum* or *proof by contradiction* or *modus tollens*, to science.

[12] Popper (1963) (chap. 1), controversially purports to answer Hume's "Problem of Induction". He argues that, given man's fallibility, one cannot identify science with certain truth because certainty is a subjective, psychological state rather than an objective, logical one. Instead, he argues, science should strive for objective *truth* rather than subjective *certainty*. Theory acceptance should always be tentative whereas theory rejection can be decisive. Popper argues that this questioning attitude liberates us from fears and remorse since we no longer need to feel disgraced when our hypotheses are falsified. In fact, he argues, improvement in science depends on it.

The following simplified example illustrates the difference between falsification and verification: No number of singular observation statements of white swans, however large, can empirically verify the statement "All swans are white" but the first observation of a black swan can falsify it. If one sets out to find confirming instances of the above statement, one would not have much difficulty. However, this would only corroborate the general statement—not prove it conclusively.

According to Popper, a theory belongs to science only if it is testable.[13] And a theory is testable only if some conceivable observation can falsify or refute it. *Falsifiability, therefore, is Popper's criterion of demarcation between science and non-science.*

Scientific theories, Popper argues, should be formulated as clearly and precisely as possible so as to expose them most unambiguously to falsification.[14] Theories should also make bold claims because this increases their informative and predictive content, as well as the likelihood of proving them wrong. Highly falsifiable theories are also highly testable, Popper argues.

Popper's principle of falsificationism requires one to test one's theories rigorously, to search for one's errors actively, and to eliminate one's false theories ruthlessly. It is a process that emphasises criticism and self-scrutiny: scientific tests should be designed critically to falsify or refute theories, not to support or confirm them.

Each failed attempt at refuting a theory is considered a *corroboration* of the theory. Corroboration is a failed attempt at refutation, which is not equivalent to a successful attempt at confirmation, since confirmation aims at proving theories, whereas Popper argues that no matter how well corroborated a theory may be, it always remains a tentative approximation to the truth. Amongst the falsifiable theories that survive repeated attempts at refutation, the best corroborated one—the one with the highest degree of testability and the highest informative content which has undergone and survived the most severe tests—is chosen.

[13] Theories which cannot be empirically tested do nonetheless, according to Popper, still have meaning and can still be critically discussed. But since they cannot be falsified, it means they are outside science. These theories, according to Popper, are part of *metaphysics* (a version of which will be the topic of discussion in section 5.3 of this chapter).

[14] For example, the falsificationists would argue that one should strive to eliminate normative and imperative statements e.g. "Do A before B" and replace them with falsifiable hypotheses. Not all scientists would agree with this approach though. The constructivists, for example, would argue that it is exactly such statements that constitute science.

Popper's approach towards epistemology (in contrast to his approach towards social engineering, which will be the topic of Part III) is succinctly summarised by Miller:

> Popper *prescribes* revolutionary thinking in science because its products, imaginative new theories, are easily relinquished if mistaken. It is exactly for the same reason that he *proscribes* revolutionary activity in society. For its consequences, which are rarely possible to foresee, are almost always altogether impossible to overcome.[15]

The following section will investigate the degree to which software testing in general is endorsed by Popper's epistemology—especially his notions of falsificationism and error elimination[16]. It will be followed by an investigation into whether any of the XP values, principles and practices are aligned with these Popperian notions.

5.1.2 Falsificationism and Software Testing

This section conjectures that the susceptibility of software to testing demonstrates its falsifiability, and thus, the scientific nature of software development.[17]

Software *developers* resemble theoretical scientists since they are responsible for establishing, by careful *a priori* reasoning from the customer's requirements, an overall design or "theory" of a software system, which will guide its implementation. Moreover, software *testers* resemble experimental scientists since each test case they write is like a "scientific experiment" which attempts to falsify a part of the developer's implementation. Indeed, by being testable, logical, mathematical, rigorous, refutable and deductive, software development closely resembles physical science.

Transferring Popper's principle of falsificationism by analogy to software testing would mean that software testers should not aim to prove that software programs are error-free but should rather try to uncover as many errors in the software as they can, using test

[15] Miller (1983) (p. 12). The important point here, namely, that Popper rejects revolutionary activity in societies should be borne in mind for the discussion in section 6.1.2 of the following chapter.

[16] Note that falisificationism is more specific that error elimination since it involves the application of error elimination to the natural sciences. Both terms, however, are part of a broader, critical and negative approach to philosophy, namely *fallibilism*.

[17] Despite its apparent triviality, this point is actually important since it provides a line of demarcation between science and pseudo-science. More specifically, it shows that software engineering resembles a scientific discipline as opposed to fortune telling, for instance. Admittedly, not all software is equally amenable to testing. In fact, sometimes, testing is very difficult, especially in concurrent systems. However, this is more rare than it is common.

cases.[18] Therefore, the focus of testing should be on detecting errors (which the software developer should then correct), rather than on writing test cases to confirm that the software functions according to the requirements. After all, one can imagine that it is not too difficult to construct a test case which confirms the functionality of a specific part of a program, especially if the test case writer happens to be the same person as the developer of that program.[19] Test cases should go beyond testing the required functionality of the software system to testing unusual and unexpected events, boundary conditions and so forth. Equally important, in order to aid the software testers' process of rigorous testing, software developers should aspire to write software that better supports the software equivalent of falsification, for example, by writing loosely coupled modular software.

Coutts (2008) is one author who also traces the similarities between software testing and falsification, albeit somewhat naïvely:

> ... falsification is just as essential to software development as it is to scientific development. Whereas in science the falsification criteria [*sic*] is often used as a demarcation between science and non-science, in software the falsification criteria demarks [*sic*] the testing discipline from the analysis and development disciplines.

5.1.3 Falsificationism and Formal Verification

Popper's principle of falsificationism not only provides an ideal for rigorous software testing. It also seems to coincide with another approach in software engineering of determining whether software satisifies the expected requirements, namely, *formal verification*[20]. In particular, Popper's principle of falsificationism seems to be useful in providing a philosophical understanding of the view held by many proponents of formal verification that *no amount of software testing can prove a program to be correct.* The prominent computer scientist and

[18] Clearly, the successful compilation of a computer program means that certain errors, namely syntactic and static semantic errors, are absent provided that the compiler itself is correct. However, certain other errors—especially logical ones—cannot be detected during the compilation process. These are the types of errors that software testers aim to expose.

[19] This pitfall is commonly know as the fallacy of asserting the consequent and is all too common in the software engineering discipline, from my experience. At a large multi-national corporation where I worked, software developers often wrote test cases only to confirm their implementation. But often their implementation contained assumptions about the requirements and sometimes they even embodied misinterpretations of the requirements, which meant that the test cases failed to detect these problems. While the inadequacy of testing in this case could at least partly be attributed to time pressure, it also at least partly points to an uncritical approach of the developers.

[20] FVE (2008)

leading advocate of formal verification since the 1970s, E.W. Dijkstra, is one of many who supports this viewpoint. In fact, Dijkstra famously stated "Testing can show the presence of errors, but not their absence"[21]. Formal verification "starts with a mathematical *specification* of what a program is supposed to do and applies mathematical transformations to the specification until it is turned into a program that can be executed" (EWD, 2008). The aim of formal verification is to prove or disprove the correctness of algorithms with respect to the specification using formal methods of mathematics. In so far as attempting to *prove* correctness is concerned, this approach may seem similar to the scientific method of verification, which would imply that it is in contrast to falsificationism. However, formal verification in software differs fundamentally from verification in science since the former does not make inductive inferences from the specific to the general. Rather, it uses a deductive method in order to derive algorithms from specifications using mathematically-based techniques. Its deductive nature is indeed more similar to the method of falsificationism: the theorem prover admits her own fallibility by exposing to others the argumentation she has used to derive the proof, thereby broadening the scope for falsificationism.

Formal verification and software testing are not mutually exclusive. While the proponents of formal verification disapprove of an absolute reliance on software testing, and testing alone for ratifying software, they do not view formal verification as an alternative to software testing. Instead, the two approaches can be seen as complementary: one could begin by deriving an algorithm using the method of formal verification, then go on to implement it in software, and thereafter test the software using the deductive approach described above[22]. Of course, formal verification may not be necessary for all software projects. However, it is often beneficial for those that deal with life-critical software. Dijkstra also believes that, as a first step, exercises in formal verification are important for novice programmers before they delve into implementing software[23]. Finally, formal verification provides insights into the origination of a theory—something which Popper's theory does not address.

The following section will assess which, if any, of the XP values, principles and practices are aligned with Popper's principle of falsificationism, and his related ideas of criticism and

[21] Certain authors consider this statement of Dijkstra's to be a special case of Popper's falsifiability principle, for instance, Snelting (1997) (p. 23).

[22] This approach is similar to the approach used in traditional engineering. For instance, the civil engineer specifies the design of the bridge that is to be built using mathematics, the bridge is built, and its strength is tested. An alternative approach towards software development in which test cases are written before the software is implemented, will be described in the following section.

[23] Dijsktra (1989)

error elimination.

5.1.4 Falsificationism and the XP Methodology

This section will investigate how far the XP practices[24] of *test-first programming*, *pair programming*, *single code base*, *shared code*, *root-cause analysis*, *incremental design*, and *stories*, as well as the XP principle of *failure* and the XP value of *courage*, support and encourage the software equivalent of Popper's falsificationism.[25]

Falsificationism and the XP Practices

Test-first Programming

At the most basic level, software testing in contrast to detailed planning seems to be very Popperian. One of the fundamental characteristics which distinguishes XP from traditional software methodologies is its practice of *test-first programming*. Indeed, Beck states that "In XP, testing is as important as programming". The main focus of the test-first programming practice is on defining test cases before implementing any software. The role that software testers fulfill, therefore, is shifted to the start of the project lifecycle. Clearly, this approach eliminates the pitfall mentioned earlier, namely, of writing test cases merely to confirm the existing implementation. Instead, since test cases are always written before any implementation exists, there is a smaller chance that only confirming test cases will be created. Test-first programming also increases the likelihood of a comprehensive test case suite since test cases are always written in their entirety before any software is implemented. This is unlike the situation that usually occurs during traditional software development. Using a traditional methodology like the *Waterfall Model*, the testing phase would be the last phase in the process. Almost inevitably, this phase is shortened because projects often run late, resulting in incomplete and often uncritical testing. Writing test cases first, is also an effective way of designing a software system, since the responsibilites (and even non-functional qualities) of a particular module are described precisely and formally.

[24] Beck & Andres (2005) (p. 35) point out that the XP "practices are your hypotheses" . This statement suggests that, at a meta-level, the XP practices themselves can be tested and discarded if not found to be useful, an approach which is aligned with Popper's method of conjecture and refutation.

[25] The reader should note that the discussion in this section relates specifically to XP and to those software projects most suited to XP software development. Recognising the fact that software can be developed in a wide variety of contexts, it is conceded a priori that the discussion in this section may not be relevant to certain types of projects, for instance, those which develop life-critical software and which cannot, therefore, tolerate any known errors.

In test-first programming, once the test case suite has been defined, each of the test cases within it fail when executed since the software is yet to be written. The failure of each test case points out lacking functionality. Then, as was described in section 2.4, the required functionality is implemented in an iterative-incremental style according to the most important or urgent business requirements. This cycle is repeated until all the required functionality is implemented, and all (or at least 95% of the) test cases in the suite execute successfully. It is important to note that this development approach is not as rigid as the description above may suggest. Rather, the iterative nature of the process makes it possible for projects to adapt to changing circumstances throughout the project lifecycle. These changes will inevitably necessitate that the test cases are refined throughout the project lifecycle too.[26]

From a Popperian perspective, the practice of test-first programming can perhaps be understood as continuous attempts at falsification since the failing test cases (analogous with scientific experiments) point out that the program/design (analogous with the scientific theory or hypothesis) does not function according to the requirements (analogous with the scientific problem). In this regard, scientific theories are similar to software programs which continually evolve: it is seldom the case that a scientific theory is comprehensive from the start. Instead, scientific theories are usually refined during cycles of experimentation. So too in test-first programming, the software program (the theory of a solution to the customer's requirements) is always incomplete at the start of the project lifecycle. It is then expanded over iterations until it is mature enough to satisfy nearly all (or at least 95%) of the test cases in the test suite. An important difference between the two approaches should be noted though: the software program is not discarded when it fails to run a test case successfully as a scientific theory would be if a scientific experiment or set of experiments decisively falsifies it[27]. Instead, the program is refined or enhanced incrementally with the aim of satisfying the test cases (scientific experiments) and meeting the requirements (problem).

In reality though, scientific theories are not discarded unconditionally. Instead, often the

[26] Note that the approach towards testing described here is most effective at the unit level—when there are clean and uncomplicated interfaces between units. At the code integration level, on the other hand, where interface interactions may be unpredictably ordered or may need to be resistant to physical component failure, such testing is usually only a first step. For a description of unit and integration testing, see respectively UTE (2008) and ITE (2008).

[27] Popper has been criticised for being overly vigourous in advocating falsificationism and criticism, since a vigorous approach may lead to theories being discarded before being given a fair chance. On the other hand though, Popper argues that theories should be strengthened as much as possible, before they are criticised.

best corroborated theory is chosen despite known flaws or limitations. So too in software engineering: a software program is often accepted even if it may not perfectly satisfy the customer's requirements but is nonetheless the best alternative available. Only if certain fundamental requirements are not met and a better alternative exists, will the program be discarded or proclaimed a failure, like a convincingly refuted scientific theory. Despite certain noticeable differences between the two methods, in both cases, what seems to be held invariant is the problem: in software, the software program (theory) must change to suit the customer's requirements (problem) and, similarly, in science the hypothesis (theory) must be refined to satisfy a real-world problem. Therefore, while the XP software methodology does not coincide completely with Popper's scientific methodology, it does, at least, by placing a stronger emphasis on the importance of rigorous and critical testing than traditional methodologies, seem implicitly to support a software engineering equivalent of Popper's falsificationist approach.

Continuous Testing

A refinement of test-first programming which Beck calls *continuous testing* is an XP practice which facilitates error elimination:

> In continuous testing the tests are run on every program change... Continuous testing reduces the time to fix errors by reducing the time to discover them.[28]

This statement of Beck's places a strong emphasis on error elimination. By testing programs continuously, according to Beck, not only is the Defect Cost Increase (DCI)—*the sooner you find a defect, the cheaper it is to fix*[29]—reduced, but the number of deployed defects is also minimised.

Pair Programming

Another XP practice, called *pair programming*, where pairs of programmers sit together and co-write software, also seems to support and encourage a software engineering equivalent of Popper's falsificationism. According to Beck, pair programmers:

- keep each other on task;

[28] Beck & Andres (2005) (p. 51)
[29] Beck & Andres (2005) (p. 99)

- brainstorm refinements to the system;
- clarify ideas;
- take initiative when their partner is stuck, thus lowering frustration; and
- hold each other accountable to the team's practices.

This description shows that pair programming encourages a critical peer review process whereby programmers subject each other's software to continuous scrutiny, similar to the way scientists in a scientific community scrutinise each other's theories and subject them to critical experimentation. Pair programming also seems to encourage objectivity (or at least inter-subjectivity) due to the presence of a second programmer. Regarding error elimination in pair programming, Beck states: "Many of the social practices in XP, like pair programming, tend to reduce defects. Testing in XP is a technical activity that directly addresses defects". Beck's claim regarding the benefits of pair programming is corroborated by several independent studies, which have found that pair programming results in better code, fewer defects, better design, and better team building[30].

Johnson (2006), while not disputing the benefits of error elimination that result from pair programming, provides a critical perspective on the objectivity of pair programming. He argues that the objectivity of a broader scientific peer review process in which reviews are conducted anonymously by reviewers who are entirely independent of the material being reviewed, is not evident in pair programming since "both parties have been intimately involved in the production of the material being reviewed"[31]. While this seems to be a valid criticism, Beck would likely point out that XP encourages pair programmers to rotate every couple of hours[32], which means that the same two programmers do not often work on the same piece of software for very long[33]. In addition, the source code is open to scrutiny by other members of the XP team, as the discussion below about the XP practices—*single code base* and *shared code*—will highlight. Nonetheless, since the peer review process in XP is limited to the XP team itself, which would most likely be smaller than a scientific peer review board, and whose members are more likely to share common interests, Johnson's

[30] See, for example, Williams *et al.* (2000), Cockburn & Williams (2001) and Flor & Hutchins (1991).
[31] Johnson (2006) (p. 139)
[32] Beck & Andres (2005) (p. 43)
[33] Some may argue that this approach would lead to a lack of familiarity and expertise, with potentially negative consequences.

point still seems to be, at least partly, valid. Despite these criticisms that XP's practice of pair programming is not entirely objective, it is evident that the agreement resulting from pair programming (regarding the implementation in software of a customer's requirements) is achieved through a process emphasising criticism and error elimination (in line with Popper's critical method) rather than through mere consensus (as a Kuhnian approach[34] suggests).

Single Code Base

Yet another XP practice which encourages a software engineering equivalent of Popper's falsificationist approach, is that which recommends a *single code base*. It is similar to pair programming in terms of peer review and objectivity but the number of observers is extended to the entire XP team. The individual source code contributions made by developers are stored in *shared code* repositories and through an avowedly democratic peer review process[35], the functioning of the source code in the repositories is said to be thoroughly assessed.[36] It can be conjectured that a single code base also aids repeatability—a crucial element in all sciences, especially in software engineering—because the solution to a problem must be objectively repeatable[37]. In other words, the solution should not be dependent on any individual member of the XP team. In XP, repeatability is demonstrated by the regression test suite which is stored alongside the source code in the shared code repository. The test cases, which form part of the suite show the problem being solved and allow the problem's solution to be reproduced.

Johnson (2006), however, argues that equating regression tests with the concept of scientific reproducibility is untenable. He argues that the independence of the scientific procedure in which anonymous scientific researchers successfully replicate the results of other researchers, is missing in software regression testing because "each run of the regression tests is conducted under exactly the same circumstances as the preceding ones"[38]. While

[34]Kuhn (1970) (p. 94). Although Kuhn did not say as much, he may, however, have agreed that testing and criticism could be ways to achieve consensus.

[35]In my experience, however, even though source code is accessible to the whole team, a peer review process is uncommon.

[36]This broader peer review ethic, which is central to any scientific discipline, is also central to Open Source Software Development (OSSD) where competing designs, analogous with Popper's competing scientific hypotheses or theories, are open to extensive peer review. OSSD will be discussed in more detail in Appendix F.

[37]It is acknowledged that certain solutions are not easily repeatable, for instance, those which involve distributed systems.

[38]Johnson (2006) (pp. 139-140)

this would commonly be the case, it should be noted that nothing prevents the shared source code from being tested by a remote group of software developers in a completely independent testing environment. Thus, while the practices of *single code base* and *shared code* may not necessarily mean that XP is closer to physical science, despite containing open and testable solutions to software problems, it can at least be argued that the shared code repositories encourage criticism and promote error elimination through their openness.

Root-Cause Analysis

One of the XP corollary practices called *Root-Cause Analysis*, which Beck describes by saying: "Every time a defect is found after development, eliminate the defect and its cause", seems also to be aligned with a falsificationist approach. Furthermore, it seems that this practice aims to prevent errors from being repeated: "The goal is not just that this one defect won't recur, but that the team will never make the same kind of mistake again".[39]

Incremental Design

In line with Popper's fallibilist philosophy, the XP community accepts that people invariably make mistakes. Embracing this fact, they propose an approach towards software development called *incremental design*. This approach, which derives historically from RUP (as was described in section 2.3), is now central to all Agile methodologies. It involves making "baby steps"[40] instead of "large leaps" when implementing a software program, allowing for mistakes to be detected early and repaired, by reworking parts of the program before the errors accumulate to an unmanageable number. Not only are mistakes in the software itself easily repaired, but so too are mistakes in the design of the software. The adoption of an iterative incremental approach towards software development would mean that the software design itself emerges over time and can be changed relatively easily if necessary. Incremental design also encourages regular *feedback*—another important concept in Popper's evolutionary epistemology. Finally, by shortening the cycles of releases, the likelihood of fundamental changes to the requirements during the development of a single release is

[39] Beck & Andres (2005) (p. 64)

[40] The XP principle which encourages "baby steps" relates to certain XP practices, including *test-first programming* and *continuous integration* (in which developers are encouraged to integrate their contributions into the single code base after no more than a couple of hours). The latter practice, paired with another XP practice advocating a *ten-minute build*—which requires the whole software system to be automatically built and all of the test cases to be run in ten minutes—are said to decrease risk on projects by providing regular and continuous feedback about the status of the software.

minimised.

This iterative-incremental approach can be contrasted with *Waterfall Model*-like methodologies, which encourage comprehensive a priori design. The latter approach does not anticipate feedback to the same extent as the former since the design is intended to be largely complete before implementation begins.[41] In contrast to this approach, the iterative incremental approach of Agile methodologies is considered adaptive rather than anticipatory.[42] The Agile approach seems to be aligned with Popper's approach of error detection and elimination because the software is built in small steps, allowing designs to be reworked and errors to be rectified after each iteration. Furthermore, this approach, together with the practice which advocates *refactoring* (the improvement of the design of the software without changing its underlying functionality), suggests that it is feasible for software engineers to experiment with alternative designs because unsuccessful ones can easily be discarded. Refactoring, in particular, encourages error elimination and enables systems to start small and to grow, eliminating the need for comprehensive a priori design.

By analogy with Popper's falsificationist approach, software designs can be seen as hypotheses, which detail a potential solution to the customer's requirements. The requirements, in turn, can be seen as analogous with a scientific problem. Just as Popper argues "[o]n the scientific level, we systematically try to eliminate our false theories—we try to let our false theories die in our stead"[43], an analogy would suggest that software engineers should try to let their software designs die in their stead.

Stories

Finally, it can be argued that the practice of negotiating product features using *stories* throughout the XP project lifecycle, instead of defining fixed requirements at the start of a project, encourages critical debate. The overall product features are continuously refined

[41] The parallel between the approach of traditional software methodologists compared with the approach of scientists practising Kuhnian normal science, should be noted here. Just as physical scientists in the dominant paradigm dismiss anomalies in order to maintain the integrity of the paradigm, so too do traditional software methodologists often choose to ignore potential refutations of their software designs. This approach is, of course, in conflict with the fallibilist approach which Popper advocates. The contrasting approaches of traditional versus Agile methodologies will be discussed in greater detail in Part III of this dissertation.

[42] The emergent character of software developed using this approach is also a defining characteristic of OSSD. An analogy of the principles underlying this type of software development and those underlying Popper's notion of an *open society*, will be provided in Appendix F. It is also worth noting in this context that, in OSSD, the emergent character of software development is extended to distributed collaborations, which are formed dynamically rather than through detailed a priori planning.

[43] Magee (1986) (p. 73)

during software development in a manner similar to the way scientific hypotheses are refined and improved through falsificationism. Beck points out that XP stories are more flexible than "requirements", which carry "a connotation of absolutism and permanence, inhibitors to embracing change"[44].

Fallibilism, Falsificationism and the XP Values and Principles

For a similar reason that Popper encourages bold hypothesis formation in science, the XP community encourages risk-taking and views failure as an inevitable consequence of man's fallibility. This section investigates how far the XP value, *courage*[45], and the XP principle of accepting *failure*, are aligned with Popper's fallibilism.
Beck summarises the benefits of *courage* as:

> The courage to speak truths, pleasant or unpleasant, fosters communication and trust. The courage to discard failing solutions and seek new ones encourages simplicity. The courage to seek real, concrete answers creates feedback.[46]

Regarding *failure*, he advises:

> If you're having trouble succeeding, fail. Don't know which of three ways to implement a story? Try it all three ways. Even if they all fail, you'll certainly learn something valuable.[47]

The reason XP view failure as acceptable is because it usually imparts knowledge. It is for a very similar reason that Popper advocates bold, revolutionary theory formation in science: if a bold scientific theory is falsified, it provides us with a great deal of useful knowledge due to its high informative content, and if it is corroborated, it also provides us with knowledge, albeit of a tentative nature[48]. Extending this analogy to a software project as a whole, XP advocates believe that, even if a project fails, progress is still made because people learn from their mistakes[49]. They "see problems as opportunities for change"[50] and,

[44]Beck & Andres (2005) (p. 44)
[45]Of course, courage is a virtue not a value but Beck classifies it as one of the XP values.
[46]Beck & Andres (2005) (p. 21)
[47]Beck & Andres (2005) (p. 32)
[48]A historical example of such a bold scientific theory, which was corroborated, occurred in 1919 when Eddington first tested Einstein's theory using his famous eclipse experiment. See Miller (1983) (p. 122).
[49]It is not denied that this seems a somewhat näive position since, in reality, a failed software project could result in bankruptcy for a company.
[50]Beck & Andres (2005) (p. 30)

as accountable citizens, "accept responsibility for the consequences"[51] of their decisions.

The preference that XP has for risk-taking can be contrasted with the preference that process improvement methodologies, like CMM, usually have for low-risk behaviour. As DeMarco & Lister (1999) point out, when the pressure of process improvement becomes the goal, low-risk projects are usually chosen above high-risk projects. However, these low-risk projects are also usually low-benefit projects—in other words, they are projects probably not worth pursuing at all (p. 192). An analogy between high-risk projects and bold (i.e. highly falsifiable) scientific hypotheses, in this context, suggests that both potentially yield a greater benefit to the community than their opposites.

Summary

In summary, this section has shown, using Popper's principle of falsificationism as a criterion of demarcation, that software engineering is a scientific discipline in so far as the software it produces is amenable to testing. In addition, this section has shown that several XP values, principles and practices have a strong affinity with Popper's critical method.

In preparation for the discussion in the following section, which focuses on Popper's four-stage *evolutionary theory of knowledge*, the relation between falsificationism and the latter theory will be briefly discussed here. Falsificationism is a principle which guides the process of theory formation in science: it encourages scientists to find experimental conditions under which their theories fail. As will become evident in the following section, falsificationism is used mainly during stage three of Popper's four-stage model, namely, the stage of *error elimination*. Therefore, Popper's *evolutionary theory of knowledge*, a model which describes the entire process of Popper's epistemology, can be seen as encompassing falsificationism. For this reason, the ideas underlying Popper's principle of *falsificationism* and his *evolutionary theory of knowledge* overlap somewhat and are mutually interdependent. In order to avoid repetition, the reader should keep the discussion in this section in mind when the stage of error elimination in Popper's four-stage model is discussed in the following section.

[51] Beck & Andres (2005) (p. 4)

5.2 Evolutionary Theory of Knowledge

In this section, a detailed analogy will be drawn between Popper's evolutionary epistemology and the approach of *iterative-incremental* software development advocated by the Agile software methodologists.[52] It will be investigated how far this analogy is appropriate, and the limitations of this analogy will also be highlighted. To the best of my knowledge, no one from either the Agile community or the broader software engineering community has made such a comparison. Therefore, the aim of this section is to systematically investigate the similarities between a software methodology (in this case, XP) and a scientific methodology (in this case, Popper's evolutionary theory of knowledge). To prepare the reader for this analogy, a brief description of Popper's evolutionary theory of knowledge will first be provided. The remainder of the section will then investigate whether Popper's evolutionary theory of knowledge is suitable for explaining cumulative changes that occur during everyday Agile software development.

5.2.1 Popper's Evolutionary Theory of Knowledge Described

According to Popper (1999), all knowledge, of which scientific knowledge is the exemplar, begins with problem-solving. Popper uses the following four-stage model[53] to represent the way scientific knowledge advances:

$$P_1 \rightarrow TS \rightarrow EE \rightarrow P_2 \ . \tag{5.1}$$

P_1 is the initial *problem*[54], TS the proposed *trial solution*, EE the process of *error elimination* applied to the trial solution, and P_2 the resulting *solution with new problems*[55].

[52] It could be argued that it is not a *mere* analogy but rather a *homology* in the sense that Popper used the term when comparing animals' organs to human organs: "Such thinking in homologies is a *premise* of evolutionary thinking" and "Such thinking in homologies must be extended to our knowledge, to the acquisition of knowledge and to knowledge in general" (Popper, 1999) (p. 50). In this context, software development methodology can be considered homologous to scientific methodology, and can arguably be said to have emerged from the latter.

[53] Popper points out that there is a superficial resemblance between his own evolutionary schema and Hegel's dialectic but notes that there is a fundamental difference: "My schema works through error-elimination, and on the scientific level through conscious criticism under the regulative idea of the search for truth... Hegel, on the other hand, is a relativist. He does not see our task as the search for contradictions, with the aim of eliminating them, for he thinks that contradictions are as good as (or better than) non-contradictory theoretical systems..." (Miller, 1983) (p. 77).

[54] The initial problem is derived from an existing theory instead of generalising from observed instances, as the inductivist method proposes.

[55] (Miller, 1983) (p. 70)

This model holds that complex structures can only be created and changed in stages, through a critical feedback process of successive adjustments. It is an evolutionary model concerned with continuous developments over time as opposed to full-scale development based on an unchanging model or blueprint.[56] According to this model, scientific knowledge is perpetually growing. There is no state of equilibrium[57] because the tentative adoption of a solution to one problem (P_1), invariably exposes a set of new problems (P_2) which, in turn, become input (P_1) to a subsequent cycle of the model[58]. Popper regards the way in which scientific knowledge continuously grows and changes, a result of our perpetual attempts to solve problems.

It is important to understand that although the problem-solving process in Popper's model is continually iterated, it is **not** merely cyclical[59] because P_2 is always different from P_1 and is always a closer approximation to the truth. P_2 also feeds back into the next cycle of problem-solving. An important aspect to note about the order of the four stages in Popper's model is their conflict with the order of the stages of the traditional inductivist method of science. The latter model holds that knowledge of the external world is derived from our sense perceptions. Therefore, the inductive method always begins with observations, and ends in a generalisation[60]. Popper's model, on the other hand, always begins with a problem from the real world. His model emphasises criticism and error elimination of hypotheses whereas the inductivist model emphasises confirmation or verification of hypotheses.

Popper points out that his evolutionary theory of knowledge can be applied not only to the progress of scientific knowledge but also to the development of knowledge in the individual—both human and non-human. He argues that each cycle is to some degree informed by memories of what survived prior cycles. Therefore, if one traces this process backwards, he argues, one finds unconscious expectations that are inborn. This explains how Popper's evolutionary theory of knowledge merges with a theory of biological evolution[61],

[56] Popper applies this idea of "continuous developments over time" to his philosophy of ethics too, as section 6 of the following chapter will describe.

[57] Contra Kuhn's homeostatic model (Fuller, 2003) (p. 83).

[58] The cyclic nature of Popper's model can also be compared to Boehm's Spiral Model, which was described earlier in section 2.3.

[59] Unlike Kuhn's *merely* cyclical model consisting of an endless cycle of normal and revolutionary phases (Fuller, 2003) (p. 82).

[60] Popper (1999) (p. 6)

[61] In fact, the third stage in Popper's model, error elimination, is analogous with natural selection, according to Popper. Popper used the term "homology" to indicate that we can never make an absolutely fresh start since before we are even conscious of our existence, we have been influenced for a considerable time by our relationship to other individuals. In other words, we inherit the *whole* past from our culture and

and shows that it is much broader than scientific method. What distinguishes the evolution of scientific knowledge from biological evolution, according to Popper, is the conscious application by human beings of the critical method.

In the previous section, it was said that Popper prescribes *revolutionary* thinking in science. The reader may wonder how this can be reconciled with Popper's *evolutionary theory of knowledge*, as described above. The following will attempt to clarify this apparent contradiction. It is during the stages of *TS* and *EE* that Popper encourages revolutionary activity: in *TS*, through the formation of bold and imaginative hypotheses, and in *EE*, through rigorous error elimination using critical experiments. Popper's four-stage model as a whole is, nonetheless, *evolutionary* since the output of one cycle always becomes the input to another, and this cyclic process is repeated indefinitely.

It may be informative to attempt to compare Popper's evolutionary model to Kuhn's model of scientific revolutions. We could, firstly, attempt to compare Kuhnian normal science to Popper's four-stage model as a whole. However, these two approaches differ starkly since scientists practising normal science in a particular paradigm uncritically go about puzzle-solving—they do not attempt to refute theories or the paradigm[62]—whereas scientists using Popper's method aim to refute their theories using a revolutionary approach of bold hypothesis formation and critical error elimination[63]. We could, secondly, attempt to compare a paradigm shift to the final stage in Popper's model. However, in Popper's model, the output of a cycle feeds back into the following cycle, whereas in Kuhn's model, the new paradigm is a complete break from the old—so much so that the old and new paradigms are incompatible[64]. This is why Kuhn's model is considered revolutionary whereas Popper's, evolutionary. From this brief comparison, it is evident that the two models are two largely incompatible theories of scientific knowledge. Even the terminology of the two models seems

tradition. Consequently, Popper argues that scientists should study the problem situation of the day. This approach will be shown later in this section to be similar to the approach advocated by the XP community, namely, that requirements should only be implemented in software if there is a high degree of confidence today that they will be needed.

[62]Some may think that attempts at refutation are not necessary in Kuhn's model if the paradigm is satisfactory in explaining reality. However, according to Kuhn, anomalies do occur and are ignored, which suggests that the paradigm is not satisfactory under all circumstances.

[63]Eddington's famous eclipse experiment of 1919, which was mentioned in the previous section, is a case in point. Popper contends that Eddington's experiment was a dramatic corroboration of Einstein's theory of relativity as a result of a critical attempt to refute it.

[64]The inaccuracy of this conjecture of Kuhn's is evident if one considers, for example, that Einstein's relativity theory "contains" Newton's theory as a special case, even though the former theory triggered a "paradigm shift".

to be largely incompatible. For instance, Popper uses "problem-solving" whereas Kuhn uses "puzzle-solving", and "counter-instances" in contrast to "anomalies".

5.2.2 Popper's Evolutionary Theory of Knowledge and XP

Popper's evolutionary theory of knowledge has been applied to various fields of human inquiry apart from science, including music, art, economics, logic and learning.[65] In fact, virtually all forms of organic development, and many forms of social development, can be understood in this way. The aim of this section is to investigate whether this claim is true too for software development. In particular, an analogy will be drawn between Popper's four-stage model and the *incremental development* approach of the XP methodology[66]. It will be investigated whether Popper's model is suited to explaining the scientific nature of XP's emphasis on flexibility in the face of change. It is important to keep in mind throughout this section that XP's avowedly flexible methodology is a self-consciously radical departure from traditional *Waterfall-like* methodologies, whose perfectionist, blueprint approach to software development arguably inhibits change.[67]

In contrast to traditional methodologists, XP methodologists believe that software development requires short and continuous feedback cycles[68] in order to control unpredictability and allow for flexibility in the face of change. Consequently, XP methodologists advocate *incremental design* as their approach towards software development. The incremental design process, according to Beck, is a way to deliver functionality early and continue delivering functionality regularly throughout the life cycle of the project. This section will investigate whether Popper's model of epistemological change is more useful in understanding Agile's incremental design process than Kuhn's more monolithic model[69].

Section 2.4 described in detail the typical activities that occur during one iteration of an

[65] In section 7.1 of the following chapter, I will attempt to apply it to another field—architecture—when I investigate the parallels between Popper's theory of knowledge and the approach towards architecture espoused by Christopher Alexander.

[66] Note that the incremental development approach has deep roots that long precede the Agile movement. The earliest reference to this development approach was made by Royce (1970).

[67] In the following chapter the two types of software methodology, Agile and traditional, will be compared, respectively, to Popper's ideas concerning piecemeal and utopian social engineering. These two divergent approaches towards software development can also be contrasted respectively with Popper's model of problem solving and Kuhn's model of paradigm-based research.

[68] The reader is referred back to Figure 2.5 in section 2.3 as a reminder of the XP life cycle.

[69] Monolithic means, in this context, that the activities of scientists' who have holistically adopted a paradigm, are completely governed by that paradigm and there is no scope for changing the framework provided by the paradigm. Change within the paradigm is mere articulation and puzzle-solving.

XP project, and the previous section discussed several XP practices that are used within an iteration. The focus in this section, therefore, will **not** be on the particular XP practices but rather, at a higher level, on the entire XP software development process, which constitutes multiple iterations over the life cycle of a single software project.

It is conjectured that the small-step software development iterations of Agile and XP methodologies can be described in terms of the cycles of Popper's evolutionary model. Taken together, the four stages in Popper's model can be interpreted, in terms of XP software development, as a single development iteration within the XP project life cycle: an existing solution P_2 (analogous with either a previously released version of the software or with a part of a software solution which is currently under development) from a previous iteration does not meet all requirements (for example, errors are detected in a previously released software version, the customer has new requirements, or there is functionality still outstanding for a solution currently under development) and becomes, thus, the input to a new development iteration, analogous with P_1. P_1 represents the specific requirement(s) (chosen from the outstanding requirements/stories) that the software developer should implement and the software tester should test in the current iteration. *TS* can then be seen as analogous with the software solution, including the design, proposed by the developer to meet the requirement(s) of the iteration. *EE* can be considered parallel with the attempt, by the software tester, to eliminate errors by writing critical test cases. Finally, P_2 is analogous with the new version of the software, which has been tested to ensure that it incorporates the new subset of customer requirements and/or solves the errors that were detected in P_1. The unanticipated problems or requirements that P_2 may give rise to in future, can then be addressed in a future iteration. This iterative process naturally results in small new increments of functionality, released to the customer as new versions of software. Thus, in much the same way that scientific knowledge advances through this evolutionary process, software approximates more and more closely the required solution over repeated iterations.[70]

It was highlighted in the previous section that the XP practice of *test-first programming*,

[70]Note that while this analogy has been drawn on a small scale between a single software development iteration and a single cycle of Popper's model, the analogy can equally well be extended to a larger scale between the whole XP *system metaphor*—the shared story of how it is envisaged the software system will work based on all known requirements—and the scientist's overall theory. On a small scale, whereas software requirements for a particular iteration are comparable with scientific hypotheses, on a larger scale, the entire set of all known requirements are comparable with the scientist's overall theory (which continuously evolves as the scientist creatively forms and tests new hypotheses).

or what Beck calls "writing software to test cases" distinguishes the XP methodology from other Agile methodologies, and especially traditional methodologies. Therefore, it could be said that the above analogy was drawn more generally between Popper's model and Agile software methodologies. An analogy between Popper's model and XP, in particular, would perhaps require an additional stage, let's say EE_1, before the stage of TS in order to accommodate the practice of *test-first programming*. In other words, Popper's model could be extended as follows:

$$P_1 \rightarrow EE_1 \rightarrow TS \rightarrow EE_2 \rightarrow P_2 \ . \tag{5.2}$$

where P_1 represents the XP *stories* chosen for the iteration, EE_1 represents the test cases written as part of test-first programming, TS represents the software implemented to fulfill the requirements of the iteration, EE_2 represents the execution of the test cases against the newly implemented software, and P_2 represents a new version of the software (containing the functionality implemented and tested during this iteration and all previous ones). This extension to Popper's original model need not necessarily invalidate the analogy. Perhaps, if we consider that a new scientific theory (P_1) can logically be tested against existing experiments used to test a previous theory (P_0), then EE_1 can be considered to be these existing experiments. Interpreted in this way, the extended model seems to provide a reasonable fit.[71]

In addition to the XP values, principles and practices that were shown in the previous section to be aligned with both Popper's principle of falsificationism and his more general

[71] It is interesting to note that a similar iterative development method for technology development had been devised in 1963 already, almost 40 years prior to the Agile movement, by the *Center for the Study of Democratic Institutions* in Santa Barbara, California. According to an article entitled "Maieutische Systemanalyse" in Krauch (1994), the method was termed the "Maieutic Cycle" and focused, in a similar way to the Agile methodologies, on the involvement of end-users and various other stakeholders. The term "Maieutic" refers to Socrates' activity as a "midwife" during the "birth" of knowledge. The idea is thus that the technologists are only "helpers" in the process, rather than the rulers. The Maieutic Cycle, which can be iterated as often as needed until all stakeholders are satisfied, consists of the following phases:

1.
   - Initial Problem
   - Differentiation
   - Experience and Understanding
   - Evaluation and Choice
   - Model or Prototype
   - Justification and Consensus
   - Experiments and Tests
2. If satisfied, produce artefact. Otherwise, go back to 1.

approach of error elimination, the remainder of this section will investigate several additional XP values and principles, which may be aligned with Popper's theory of evolutionary knowledge.

Firstly, XP's view of problem solving, namely, "to see problems as opportunities for change"[72] is embodied in its principle called *opportunity*. Beck argues that "To reach excellence, problems need to turn into opportunities for learning and improvement, not just survival"[73]. This description suggests that Beck's view of problem-solving is, in one important respect at least, aligned with Popper's view that "all life is problem solving".[74] Popper's view can be explained by the final stage in his four-stage model which continuously gives rise to new problems. These new problems are important in science because they continuously provide opportunities for the growth of scientific knowledge through the improvement and refinement of hypotheses. So too do problems in software development provide opportunities for learning and improvement. In fact, Beck argues that XP's main objective is *software development through improvement*. In Beck's own words:

> The cycle is to do the best you can today, striving for the awareness and understanding necessary to do better tomorrow. It doesn't mean waiting for perfection in order to begin... The actual design will never be a perfect reflection of the ideal, but you can strive daily to bring the two closer.[75]

The words "perfection" and "ideal" that Beck uses here when contrasting the XP approach of *software development through improvement* with the opposite approach of "waiting for perfection to begin", are Platonic concepts, many of which Popper (1966) criticises in detail. It should be noted that these terms are commonly associated with the approach of comprehensive, blueprint design advocated by traditional software methodologists. Blueprint plans, by their very nature, are inflexible and seldom open to change. Therefore, they are required to be largely "perfect", or at least as comprehensive as possible, such that they would not need to change regularly. The approach towards problem-solving, which both Beck and Popper advocate, is, in my experience, quite uncommon in industry. For example, in the large transnational software company I worked for, it was not unusual to find software

[72] Beck & Andres (2005) (p. 30)
[73] Beck & Andres (2005) (p. 31)
[74] Popper (1999) (p. 100)
[75] Beck & Andres (2005) (p. 28)

engineers who ignored problems or looked for some means of explaining them away[76].

Related to XP's principle of *opportunity*, is another XP principle called *improvement*[77]. Beck applies this principle to different aspects of XP software development when he states: "There is no perfect process. There is no perfect design[78]. There are no perfect stories[79]. You can, however, perfect your process, your design, and your stories". In the context of this dissertation, "process" corresponds with software methodology, "stories" are analogous with the features desired by the customer, and "design" is analogous with the developer's proposed solution to the "stories". Regarding "design" and "stories", the respective analogies between the developer's proposed software design and a scientist's hypotheses, on the one hand, and the stories and a scientific problem, on the other, has been pointed out already. It would seem fair to say, therefore, that in both XP software development and in Popper's evolutionary theory of knowledge, both the "design" and the "stories" are targets for improvement over the iterations. Regarding "process", the analogy between a software methodology (the XP methodology as a whole) and a scientific methodology (Popper's entire four-stage model in this case), may require further explanation. By applying the principle of *improvement* to the software methodology itself, Beck seems to be working at a meta-level. Neither during XP's iterative software development approach nor during Popper's four-stage model is the methodology itself a target for improvement. However, one can imagine that a change in a practice within a methodology, may result in a change to the methodology itself. Indeed, in the second edition of *Extreme Programming Explained*, Beck & Andres (2005) advocate that the XP practices be adopted incrementally, as required. This means that, over the life cycle of the project, the methodology itself may change as new practices are adopted. Similarly, in Popper's model, there may be changes

[76] This seems to confirm the Kuhnian conception of scientific activity.

[77] Related to this principle is the XP principle called *flow*, which suggests that teams should "deploy smaller increments of value ever more frequently".

[78] The link between improvement, design and XP's iterative incremental development approach is succinctly summarised by Beck: "Incremental design puts improvement to work by refining the design of the system" (Beck & Andres, 2005) (p. 28). The benefits of an evolving design/architecture are summarised in the *Agile Manifesto* as "An architecture that grows in steps can follow the changing knowledge of the team and the changing wishes of the user community".

[79] As pointed out in the previous section, XP stories are roughly equivalent to features. Beck explicitly argues that stories are not equivalent to requirements (in the traditional sense of the term) since the latter implies "something mandatory or obligatory". Stories are much more flexible than requirements, according to Beck, and can be negotiated throughout the XP life cycle. The earlier analogy, where P_1 is viewed as parallel to the selected XP stories for the iteration, seems to provide a good fit since the evolving nature of scientific hypotheses corresponds with the evolving nature of the features requested by the customer, which are presented as stories.

to the practices within the stages of the model—for instance, when an improved approach towards testing is discovered. This new testing approach may change the methodology itself because *EE* would have changed. However, the change will be isolated to one or more of the phases and will not likely change the methodology more fundamentally (for instance, by changing the order of the phases). Beck's advocacy, in the second edition of his book, that the XP values, principles and practices be adopted incrementally, also indicates that the XP methodology need not completely replace an existing methodology. Rather, certain XP values, principles and practices, which are deemed beneficial, may be added incrementally to those of an existing methodology. This approach to the adoption of XP, differs radically from the approach advocated in the first edition of *Extreme Programming Explained*, which entailed that the XP methodology, including all of its values, principles and practices, be adopted *holistically*. Whereas the previous chapter showed that the approach advocated in the first edition towards the adoption of XP, can be described in terms of Kuhn's ideas of "paradigm shift" and "scientific revolution", this chapter conjectures—based on the above analogy—that the approach advocated in the second edition, towards the adoption of XP, can be described better in terms of Popper's "evolutionary epistemology".

Feedback is another central aspect to both Popper's model and XP's model. The XP value of *feedback* is driven by XP's willingness to embrace *change*. As Beck states "Change is inevitable, but change creates the need for feedback". In XP, feedback can originate from many different sources, for example, tacit knowledge transfer between pair programmers, compiler errors during a ten-minute build, test case failures, new customer requirements, defect reports, and so on. The iterative-incremental development of software, and the release of small increments of new functionality, ensures that feedback is received regularly. Being flexible in order to accommodate the changes that invariably result from this feedback, is a defining characteristic of XP (and, in fact, all Agile methodologies). Feedback is also important in Popper's model since scientists rely on the feedback from experiments to improve their hypotheses. Equally important is the feedback from one cycle to the next (from P_2 to P_1) which makes possible the evolutionary development of knowledge in Popper's model. Feedback in XP is encouraged by another two XP values, namely, *communication* and *simplicity*. The importance of communication to feedback is self-evident. Simplicity facilitates feedback because, according to Beck: "the simpler the system, the easier it is

to get feedback about it"[80]. Finally, the XP principle of *reflection*, according to Beck, encourages XP teams to "think about *how* they are working and *why* they are working. They analyse why they succeeded or failed. They don't try to hide their mistakes, but expose them and learn from them". Reflection clearly encourages feedback but, just as importantly, it encourages error elimination through a process similar to Popper's falsificationism.

To conclude this section then, it seems fair to say that certain XP principles and values seem to correspond well with Popper's evolutionary epistemology, and although a point-by-point assessment has not been made, the XP methodology does not appear to correspond closely, at this small-scale level, with Kuhn's epistemology of "scientific revolutions".

[80]Beck & Andres (2005) (p. 20)

5.3 Metaphysics

The discussion in this section relates to *metaphysics*, "the branch of philosophy that deals with first principles, esp. of being and knowing"[81]. The aim of this section is to investigate whether the Agile and XP methodologists' reliance on *tacit* or *subjective* knowledge[82], does not conflict with this dissertation's aligning them with Popper's philosophy since Popper's *three worlds metaphysical model* emphasises the supremacy of *objective* rather than subjective knowledge. It will also be investigated how an ontology of software can fit into a Popperian framework.

In the previous section, it was shown that the iterative-incremental software development approach of Agile and XP software methodologies corresponds well by analogy with Popper's four-stage model of evolutionary epistemology. Cockburn, one of the creators of the *Agile Manifesto* (Beck *et al.*, 2001a), and a leading advocate of Agile software development, believes that the regular delivery of software, which results from the iterative-incremental development approach of dividing projects into smaller sub-projects, is one of two important ways that provides projects with agility[83]. The other important way, according to Cockburn, is by "replacing some of their written documents with enhanced informal communication among team members, shifting the group's organizational memory from external to tacit knowledge"[84]. *External* or *explicit* knowledge, on the one hand, is "knowledge that has been or can be articulated, codified, and stored in certain media. It can be readily transmitted to others. The most common forms of explicit knowledge are manuals, documents and procedures" (EKN, 2008). *Tacit* knowledge, on the other hand, is a form of knowledge that people carry in their minds, one which is often difficult, and sometimes even impossible to articulate. Tacit knowledge is a personal or *subjective* form of knowledge presupposing an often unarticulated and subconscious background of paradigms, cultures, beliefs, and concrete know-how, such as crafts and skills. According to Cockburn:

> All project teams rely on tacit knowledge, usually to a far greater extent than they suspect. Agile methodologies, though, place a deliberate reliance on tacit knowledge. Through the use of informal communication channels, such as colocated teams, pair programming, or daily stand-up meetings, they demand that

[81] (Makins *et al.*, 1991) (p. 982)
[82] See, for example, Boehm (2002), Chau *et al.* (2003), and Cockburn & Highsmith (2001).
[83] Cockburn (2002b) (p. 6)
[84] Cockburn (2002b) (p. 6)

> the people on the team keep each other abreast of ongoing changes to the plan, the requirements, the design, the code.[85]

The concept of *tacit knowing* (rather than tacit knowledge) was first introduced by the 20^{th} century British polymath, *Michael Polanyi*. Polanyi did not speak about a form of knowledge per se, but rather about a process—hence *tacit knowing*[86]. His phrase was, however, later used by two leading theorists in the Knowledge Management (KM) discipline, Nonaka and Takeuchi[87], to name *tacit knowledge* (TKN, 2008).[88] It is of importance to note that the Knowledge Management (KM) discipline is a distinct discipline from the software engineering discipline[89]. However, it likely that the authors of the Agile software methodologies may well have taken cognisance of the KM discipline, especially since both KM and Agile methodologies came to the fore at more or less the same time, in the early 2000s. Indeed, in the case of XP, Beck & Andres (2005) explicitly acknowledge Nonaka and Takeuchi in the annotated bibliography of their book, which shows that they were aware of the ideas of these KM theorists. Indeed, the aim of many of the XP values, principles and practices also coincides closely with the aim of KM, which is to devise ways of maximally leveraging knowledge in the subjective minds of people. Despite the links between XP and KM, this section will not make any statements about the KM discipline, as such. Instead the focus is solely on investigating the supposed reliance, as Cockburn's quote above points out, of the Agile methodologists on tacit or subjective knowledge, and whether this conflicts with

[85] Cockburn (2002b) (p. 7)

[86] It is both interesting and significant to this dissertation that Polanyi's work was made known to a larger audience by being quoted by Kuhn (1962) (p. 5). Kuhn, himself, explicitly cites Polanyi's *Personal Knowledge* and agrees with his argument that knowledge is acquired through practice and can often not be explicitly articulated. (Kuhn, 1970) (p. 44)

[87] Knowledge Management is defined by Nonaka & Takeuchi (1995) (p. viii) as "the capability of a company as a whole to create new knowledge, disseminate it throughout the organization, and embody it in products, services and systems".

[88] Nonaka and Takeuchi explicitly acknowledge Polanyi's writing as their source for the concept of *tacit knowledge* (Gourlay, 2002). However, some argue that Nonaka and Takeuchi do not use Polanyi's concept in the way he intended. See, for example, Pauleen (2007) and TKN (2008). In particular, it has been said that Polanyi's famous aphorism "We know more than we can tell" (Polanyi, 1967) (p. 4), has been misinterpreted by Nonaka and Takeuchi to mean that knowledge is of the two types—explicit and tacit—as described above. Polanyi, while believing that all knowledge comprises a tacit dimension, does not discuss a tacit type of knowledge, but rather writes of "tacit knowing". Nor does he believe in explicit or objective knowledge independent of a knowing subject since, according to him, all knowledge has an indispensable personal component.

[89] The two disciplines are not necessarily mutually exclusive though. Indeed, as pointed out in section 3.3.2, the KM discipline is often supported by the products that software engineers create and, in turn, organisations which produce software artefacts or which have many employees who require access to information resources, usually adopt some type of framework for managing the vast amounts of knowledge in the organisation.

Popper's belief in the supremacy of *objective* knowledge, since this dissertation has aligned XP with Popper's philosophy.

In order to investigate this apparent conflict, Polanyi's model of "tacit knowing" will be compared to Popper's three worlds model, firstly, because an emphasis on *subjective* knowledge is what both Polanyi's concept of "tacit knowing" and the concept of "tacit knowledge" used by Agile methodologists share in common, and secondly, because Popper (1959) specifically raises his objections against Polanyi's ideas. The focus, therefore, will be on comparing Polanyi's emphasis on *subjective* knowledge to Popper's emphasis on *objective* knowledge. If this comparison shows that Popper's model is fundamentally incompatible with Polanyi's model, it would imply, too, that Popper's model is incompatible with Agile's reliance on tacit or subjective knowledge.

This section contains three main parts. Firstly, Popper's three worlds metaphysical model will be contrasted with Polanyi's model of "tacit knowing". The discussion will highlight the importance Popper's model attributes to objective knowledge, in contrast to the importance Polanyi's model attributes to subjective knowledge. Secondly, it will be investigated to what extent the XP values, principles and practices rely on tacit knowledge, and whether they acknowledge the importance of objective knowledge under any circumstances. Finally, using concepts from the philosophy of art, a *theory of the ontological*[90] *status of software* will be developed, and Popper's model will be used in an attempt to understand the *nature* of software, which is conjectured to inhabit Popper's World 3 of objective logical content.

5.3.1 Popper's Three Worlds Model and Polanyi's Model of Tacit Knowing

Popper divides existence and the products of cognition into three ontologically related domains, which he called *Worlds*[91]:

1. *World 1*, the world of physical objects or of physical states, independent of any perceptions;

2. *World 2*, the subjective world of private mental states; and

[90] Ontology is the branch of philosophy dealing with the study of *being* or *existence*, as described at the beginning of Part II of this dissertation.
[91] Miller (1983) (pp. 58-77); Popper (1999)(pp. 23-35)

3. *World 3*, the objective but intangible world of products of the human mind.

To demonstrate the difference between the three worlds, Popper uses the example of a mathematician working on a proof whose solution is yet to be discovered. If the proof is discovered today, it will belong to World 2 (in the mind of the mathematician) and, if the mathematician writes it down, it will also belong to World 1 (on a physical medium, for instance, the ink on a sheet of paper). The proof, according to Popper, would also always have existed in World 3 (as objective logical content), awaiting discovery.[92] Therefore, if the proof is not discovered for some time, it will continue to exist only in World 3. The interaction and feedback between Worlds 1, 2 and 3 are depicted in more detail in Appendix C. A real world example of the existence of World 3 can be found in geometry: for over 2000 years, Euclid was considered synonymous with geometry. However, in the 20^{th} century, alternative systems of non-Euclidean geometry such as hyperbolic and elliptic geometry, were discovered. According to Popper, these systems had always existed in World 3, awaiting discovery.[93]

In addition to the distinction between the three Worlds, Popper further distinguishes between two types of knowledge: knowledge in the *subjective sense* and knowledge in the *objective sense*. The first type of knowledge belongs to World 2—the world of subjective belief—and the second type to World 3—the world of objective knowledge.[94]

World 1 and World 2 are the familiar worlds of *matter* and *mind* respectively. World 3, however, may require further explanation. Knowledge in World 3 consists of the logical content of spoken, written or printed statements (such as critical arguments, artistic and literary works, scientific problems and ethics), which are open to public criticism and can

[92] While it may seem, as first glance, that Popper's World 3 shares much in common with Plato's world of Forms or Ideas, Plato would consider proofs and arguments a *means* to Ideal Forms whereas Popper would regard them as inhabitants of his World 3. Indeed, Popper argues: "Plato's world 3 was divine; it was unchanging and, of course, true. Thus there is a big gap between his and my world 3: my world 3 is manmade and changing. It contains not only true theories but also false ones, and especially open problems, conjectures and refutations" (Miller, 1983) (p. 74).

[93] There may seem to be a contradiction between World 3 knowledge being a *product* of the human mind and, at the same time, being open to *discovery* by human minds. To say X is a product of Y, is to say X is produced by Y whereas to say X is discovered by Y, is to say the existence of Y is not contingent on X. Popper defends the view that World 3 is the genetic product of World 2, and that it has a partly autonomous internal structure by stating: "Some good examples come from mathematics. The series of natural numbers 1, 2, 3... and so on is, I think, a product of our language. There are primitive languages that know only 1, 2 and many... But no one invented the *prime numbers*: they were *discovered* in the sequence of counting" (Popper, 1999) (pp. 26-27).

[94] Implicit knowledge, which is knowledge that one is capable of making explicit, but has not yet made explicit, would most probably overlap with both World 2 and World 3 knowledge, although Popper never spoke of implicit knowledge per se.

be disputed. Most importantly, in the case of science, World 3 knowledge can be experimentally tested and objectively repeated. Knowledge in World 2, by contrast, is private and barely criticizable. This is the type of knowledge studied by traditional epistemology. According to Popper, the fact that knowledge in World 3 is objective and open to public scrutiny, establishes the superiority of knowledge in World 3 over knowledge in World 2 since, even though both types of knowledge are conjectural, objective World 3 knowledge is more important to the growth of scientific knowledge than subjective World 2 knowledge. Popper's belief in the conjectural nature of all knowledge can be understood by considering that knowledge often has the character of expectations, which, in turn, usually have the character of hypotheses. As was shown in section 5.1, Popper always regards hypotheses as tentative and conjectural, which implies their uncertainty. Knowledge, therefore, is also deemed by Popper to be conjectural.

A final unique property of World 3 is the existence of certain objects in it alone. An example may help to clarify this statement: the fact that *calculus* was discovered independently by Newton and Leibniz shows that it was not merely created by a subjective mind but that it pre-existed in World 3, awaiting discovery. In terms of World 2, Newton made the discovery first, but in terms of World 1, Leibniz was the first to publish the discovery in physical form.

Now, in an attempt to assess the relevance of Popper's three worlds model to Agile and XP software development, it is important to note again that by relying on tacit knowledge, the Agile and XP software methodologists have focused much of their attention on subjective knowledge, which Popper places in World 2. Popper's criticism of a reliance predominantly on subjective knowledge is best understood by comparing it to a model which emphasises subjective knowledge. Polanyi model will be chosen for such a comparison, firstly, since Popper explicitly raised his objections to Polanyi's model, and secondly, due to Polanyi's link to the concept of tacit knowledge, which the Agile methodologists use.

By emphasising World 2 knowledge, Polanyi's model addresses only the first type of knowledge identified by Popper, namely, knowledge in the *subjective sense*. The reason Polanyi emphasises subjective knowledge can be understood by considering his view of *truth*. According to Popper (1959) (p. 60), many sociologists do not believe in objective truth, but think of truth as a sociological concept:

> Even a former scientist such as the late Michael Polanyi thought that truth

> was what the experts *believe* to be true—or, at least, the great majority of the experts. But in all sciences, the experts are sometimes mistaken[95]. Whenever there is a breakthrough, a really important new discovery, this means that the experts have been proved wrong, and that the facts, the objective facts, were different from what the experts expected them to be. (Admittedly, a breakthrough is not a frequent event.)

Polanyi's understanding of truth—as what the experts subjectively believe to be true—means that he emphasises subjective knowledge (and subjective truth) above objective knowledge (and objective truth).[96]

Popper criticises Polanyi's view of truth by pointing out that human knowledge is fallible:

> I do not know of any creative scientist who has made no mistakes... Not only are all animals fallible, but also all men. So there are experts, but no authorities...[97]

This is why Popper believes in the superiority of objective knowledge. He argues that it can only be on the basis of objective tests and criticism of hypotheses that scientists come to an agreement about truth, or rather to a closer approximation to the truth. If knowledge is held instead subjectively in the minds of scientists, not only is the knowledge inaccessible to scrutiny, but the scientists tend to become unquestionable authorities.[98]

The fallibility of human knowledge, Popper argues, also necessitates the rigorous questioning of one's knowledge and assumptions. As a result of this questioning process, new knowledge is produced[99]. It is, therefore, crucial to any endeavour which involves knowledge, software development included, that conditions for openness in knowledge processing are created and maintained.

The following section will investigate to what extent Agile and XP software methodologists rely on subjective or tacit knowledge. It will then question whether their reliance on subjective knowledge implies that Popper's belief in the superiority of objective knowledge,

[95] For instance, all the experts agreed for 1500 years that the earth was the centre of the universe. Therefore, it is clear that one cannot rely merely on the subjective knowledge of experts for truth.

[96] Similarly to Polanyi, another well-known philosopher who is considered a constructivist—Paul Feyerabend—rejects objective knowledge. He believes that the laws of nature are a social construct. (Feyerabend, 1987) (pp. 60-61, pp. 88-89)

[97] Popper (1959) (pp. 60-61)

[98] It is important to draw a distinction between being *in* authority, such as a judge, and being *an* authority, such as an expert on some subject or in a discipline. A scientist is *an* authority and as such may be challenged concerning knowledge claims.

[99] This process is inextricably linked to Popper's evolutionary epistemology, which was described in the previous section.

is at odds with Agile methodologies, or whether a consideration of the importance of objective, World 3 knowledge may nonetheless be partly applicable and perhaps even provide a more comprehensive understanding of the way in which Agile and XP methodologists create and share knowledge.

5.3.2 Popper's Three Worlds Model and Agile Software Methodologies

It is conjectured in this section that Popper's identification of three distinct worlds of knowledge can provide a basis for understanding the different approaches that software methodologists adopt when creating and sharing knowledge. For instance, advocates of *process methodologies* such as the Capability Maturity Model (CMM), who emphasise the importance of documentation, consider both World 1 and World 3 knowledge but perhaps place too little emphasis on World 2 knowledge. Their reliance on unchanging software blueprints or plans, however, suggests that they fail to recognise the dynamic nature of knowledge in World 3. Indeed, their view of plans can perhaps be regarded as similar to Platonic Forms in that the plans are static and unchanging, and should not be deviated from. This section specifically investigates whether the way in which Agile software methodologists, and XP methodologists in particular, create and share knowledge is adequately explained by Polanyi's model, or whether a more balanced explanation that acknowledges the importance of both subjective and objective knowledge would be of some value.

Knowledge creation and sharing is a crucial part of successful Agile software development, as it is for most other types of software development. However, what distinguishes the way Agile software methodologists create and share knowledge from others, is the way in which knowledge sharing is achieved. As Cockburn points out, Agile teams derive much of their agility by relying on the subjective knowledge embodied in the team, rather than on explicitly writing the knowledge down in physical form. Their reliance on subjective knowledge is not surprising considering their frequent use of informal communication channels such as self-organising teams, customer collaboration, face-to-face conversation and reflection[100]. These various forms of informal communication facilitate tacit knowledge transfer and foster a culture of knowledge sharing in the organisation. According to the Agile advocates, the resulting enhancements in informal communication between team members

[100]These four principles have been taken from the *Principles Behind the Agile Manifesto*, which is provided in its complete form in Appendix A.

minimises the need for documentation and for other explicit forms of knowledge. Informal communication also helps to access tacit knowledge, knowledge which cannot easily be captured—or even at all—in documentation.

With regard to XP in particular, several values, principles and practices seem to facilitate and encourage tacit knowledge transfer. The inter-dependent XP *values* of *communication* and *feedback* both encourage tacit knowledge transfer between XP team members. The XP principle of *reflection*, is another way in which the XP team members share knowledge amongst each other. There are several XP practices, many of which were discussed earlier in this chapter, which also facilitate tacit knowledge transfer, including, *sit together*[101], *whole team*, *pair programming* and *real customer involvement*[102].

In addition to these XP values, principles and practices, which show the importance that XP places on tacit or subjective knowledge, Beck argues that XP prefers communication above written documents since it satisfies the XP principle of *humanity*. This principle is based on the observation that conversation meets the basic human need for connection more readily. Indeed, Beck argues:

> Written communication is inherently more wasteful. While written communication allows you to reach a large audience, it is a one-way communication. Conversation allows for clarification, immediate feedback, brainstorming together, and other things you can't do with a document.[103]

and

> ...talking and working go together on an XP team. The hum of conversation is a sign of health. Silence is the sound of risk piling up.[104]

Extensive internal documentation of software is, according to Beck, "an example of a practice that violates mutual benefit"[105]. *Mutual benefit*, which states that *every activity should benefit all concerned* is said to be the most important XP principle and the one most difficult to adhere to. This principle, according to Beck, is violated by extensive internal

[101] Beck writes "The other lesson I took [from that experience] was how important it is to sit together, to communicate with all our senses" (Beck & Andres, 2005) (p. 38).
[102] As Beck states, "Make people whose life and business are affected by your system part of the team" (Beck & Andres, 2005) (p. 61). Tacit knowledge, therefore, is shared not only between developers on a team but also between developers and customers.
[103] It is interesting to note that Plato also preferred spoken communication to written communication.
[104] Beck & Andres (2005) (p. 79)
[105] Beck & Andres (2005) (p. 26)

documentation since it suggests that developers should slow down their current development considerably so that some unknown person in the future may be able to easier maintain that software[106]. This means that not all concerned will benefit from the activity of extensive documentation.

All these XP values, principles and practices, which support tacit knowledge transfer, seem to be sufficiently explained by Polanyi's model of tacit knowing. This suggests that Popper's model, which adds explicit World 1 knowledge and objective World 3 knowledge to Polanyi's model of subjective knowledge, is not necessary or may even contradict the way XP methodologists create and share knowledge.

However, many Agile software methodologists, **do** recognise the importance of explicit and objective knowledge in certain scenarios (Chau *et al.*, 2003). In XP, the practice of *shared code*, for example, which is collectively owned by the XP team, encourages objectivity. The source code comments, which XP methodologists write in order to compensate for the reduced amounts of documentation in manuals, are another form of explicit knowledge created by the XP team. The collective ownership of the source code as well as the comments facilitates the advancement of knowledge[107] because each team member has some *responsibility* for keeping the information up-to-date. Collective ownership of source code cannot be adequately characterised as subjective, or even inter-subjective, as in Polanyi's model, since it involves knowledge that exists outside and independently of individual team members. Instead, the relation seems to be more adequately described in terms of Popper's three worlds model: what is owned are the objective products of the team's collective endeavours, which cannot be reduced to the sum of their individual tacit knowledge contributions.

Another XP practice, *stories* (which was discussed in section 5.1), highlights the importance of explicit knowledge[108]. Related to *stories* is the practice of *informative workspace* (described earlier in section 2.4), whereby the workspace contains explicit information that is open and accessible to anyone in the organisation. The final XP practice related to explicit knowledge—*negotiated scope contract*—also highlights the necessity for explicit knowledge.

[106] Beck & Andres (2005) (p. 26)

[107] Admittedly, program source code is usually cryptic and difficult to understand unless one is a software developer skilled in the respective programming language. Program source code, therefore, can only be considered "knowledge" to those who are able to understand it.

[108] Beck writes "Give stories short names in addition to a short prose or graphical description. Write the stories on index cards and put them on a frequently-passed wall" (Beck & Andres, 2005) (p. 45).

At the end of an XP project, XP teams often write a "Rosetta Stone" document[109], which provides a brief guide for future maintainers of the software. In this case, explicit knowledge is essential since the contents of the guide could not have been communicated tacitly because it will only be needed when someone works on the software possibly far in the future.

At a meta-level, the importance of both explicit (World 1) and objective (World 3) knowledge is evident in relation to the XP values, principles and practices themselves. Beck argues that practices should be clear and objective, and that making "values explicit is important because without values, practices quickly become rote, activities performed for their own sake but lacking purpose or direction"[110].

Taking into account the aspects of XP above, which promote and support tacit knowledge transfer, as well as those that promote explicit and objective knowledge, Popper's model seems to provide a balanced framework for understanding the various forms of knowledge that Agile and XP software methodologists deal with on a day-to-day basis. Although subjective knowledge may often suffice for daily tasks within an XP team, it is not usually possible to retain as subjective knowledge all knowledge that is required to deal with all possible tasks and events. Unusual events or tasks, for example, often require ready access to a broad base of explicit knowledge. Furthermore, it is often the case that even the knowledge which is produced and agreed upon tacitly, needs to be captured explicitly[111]. Explicit knowledge is especially important when:

1. projects persist longer than the involvement of any particular individual;

2. teams consist of many members and the rapidly expanding communication channels make tacit knowledge transfer difficult;

3. team members leave a project after the main development period and their tacit knowledge is no longer available on the project;

4. employees leave the company and their tacit knowledge is no longer available within

[109]This differs from traditional software development, which requires extensive documentation before programming begins.
[110]Beck & Andres (2005) (p. 14)
[111]The Knowledge Management discipline aims, inter alia, to attempt to codify tacit knowledge and make it explicit if possible, or at least to share and transfer it. In this sense, Knowledge Management's quest has a strong objectivist emphasis, which seems quite Popperian, and anti-Polanyian, despite the KM leaders' acknowledgment of Polanyi's ideas.

the organisation;

5. the sponsors of a project require documentation about the software in the form of user manuals or design documents.

The discussion in this section on metaphysics has used Popper's three worlds model to argue that any attempt at creating and sharing knowledge that does not consider the importance of both objective knowledge and subjective knowledge, will have severe limitations. It has shown that while Polanyi's model of tacit knowing does, in many ways, explain the XP values, principles and practices which rely on tacit knowledge (for example, pair programming), it does not provide an understanding of those values, principles and practices which rely on *objective* knowledge. Popper's model, on the other hand, provides a more balanced understanding of the way Agile and XP methodologists create and share knowledge by considering both objective knowledge and subjective knowledge. An acknowledgment of the objectivity of knowledge ensures knowledge can be openly and publicly scrutinised, and decreases the likelihood of authoritarianism, to which a subjective model lends itself. Therefore, the original problem that this section aimed to investigate, namely, whether Popper's model contradicts the Agile and XP methodologists' reliance on tacit knowledge, seems to have been resolved when one considers that the Agile and XP methodologists do, under certain circumstances, acknowledge the importance of objective knowledge.

Before closing off this section, the specific branch of metaphysics called *ontology*, which deals with the study of existence, will be discussed in relation to software engineering. In particular, the question *What is the ontological status of software?* will be raised. Since it is essential, in the pursuit of knowledge, to understand the nature of the object of ones endeavours, the aforementioned question is important. In an attempt to answer this question, the following section will investigate whether software programs are inhabitants of Popper's World 3, and will also use concepts from the philosophy of art in an attempt to understand the nature of software.

5.3.3 The Ontological Status of Software

Section 2.3 provided a historical overview of how software methods have progressed over the past four decades. That section did not, however, describe what software actually *is*,

or how software can be characterised. This will be the topic of discussion in this section.[112]

So far, this dissertation has compared software development methodologies to scientific methodologies and the process of software development to the growth of knowledge in science. It has been noted that software engineering shares many features in common with scientific methodology but that in other ways, as its name suggests, it is closer to *classical* engineering. Ideally, what it shares with both is mathematical and logical rigour. Unlike the sciences, software engineering does not produce theories in order to explain aspects of nature, but rather, like classical engineering, it produces artefacts to serve practical ends. Therefore, it may be illuminating to attempt to understand the ontological status of software—the product of software engineering.

The term "software architecture" is often used to describe the structural aspects of software, relating it not only to classical engineering (the building of complex physical structures) but also to the arts (to which architecture is related). An important difference is the fact that, whereas the design of buildings is usually permanent and complete, the design of software tends to be far more prone to change and incompleteness. On the other hand, whereas the materials from which buildings are made do deteriorate and require continual maintenance, software does not decay at all, but nevertheless needs to adapt to the changing requirements of its users. It was mentioned above that engineering produces artefacts. Etymologically, "art" and "artefact" share the Latin root *ars*, meaning craftsmanship or skill. The second part of "artefact" is from the Latin *facere*, meaning "to make". Thus software and software development share features of the sciences, engineering and the arts[113]. In what follows, concepts from the philosophy of art will be used in an attempt to understand the man-made, non-physical nature of software.

Despite the term "software architecture", software—computer programs, or "code"—more closely resembles the temporal arts (music, literature) than the plastic arts (architecture, sculpture, painting, printing). Software code can be compared to the score (or notation) of a symphony or to the text of a novel or poem, which have to be interpreted and performed in time. People speak of "programming languages", all of which have to be "interpreted" by compilers, until they are translated into the zeros and ones enacted at the most basic level of machine language. A programming language resembles the rules of

[112]Parts of this section have been adapted from Northover *et al.* (2008), for which I was the main author.
[113]See Bishop (1991) for a comparison.

inference of formal logic, although, unlike an argument, a program serves a practical end. Each time a program is run (executed), this can be considered a performance. Software, therefore, is not to be identified with its code, but with the performance (execution) of the code, and successful code is judged on its actual performance.

Nonetheless, software differs in its nature and function from art. Not all art is man-made—as is evident in "found" art and aesthetic experiences of natural phenomena—whereas all code is constructed. Furthermore, while art can be considered an end-in-itself, software is a means to some further end, hence its status as a type of engineering. Also, art-works often have an expressive or symbolic or representative function whereby the artwork expresses an emotion or stands for something other than itself or provides an interpretation of some aspect of reality. In this latter function, artworks resemble scientific theories, whereas software serves a practical, non-interpretive function. Finally, unlike most art-works, software programs are seldom complete but rather are subject to constant change. Indeed, software artefacts are perhaps better named processes, not only in terms of their continual development by programmers, but also in terms of their ongoing application by users.

Another important point about software is that it is released in different versions and each new version is distributed to the end-users in plenty of identical copies. Also from this perspective—and not only from the perspective of creation or creativity—software shares something in common with artworks such as books, printings, or musical performances. For this reason, it may well be useful to consider the philosophy of art, in order to gain a deeper insight into the "essence" or "nature" of software, by analogy. The philosopher of art, Joseph Margolis, argued that artworks (programs in this context) cannot be universals, since artworks are created and can be destroyed, whereas universals cannot[114]. He used Glickman's insight that "particulars are made, types created"[115] and theorised that all artworks are tokens-of-a-type, the tokens being physically embodied and the types existing only in the form of its tokens. Types are abstract particulars and the type\token concept is distinguished from the kind\instance and set\member concepts.[116] For example, every performance of Beethoven's *Eroica* would be a token of that type, but in this case a temporal

[114]Margolis (1980) (p. 17)
[115]Margolis (1980) (p. 17)
[116]Margolis acknowledges his indebtedness to Peirce for the type\token distinction, although he also notes that he departs from Peirce's usage, which perceived types and tokens exclusively as signs.

rather than spatial token. Analogously, software programs can be considered tokens of a type existing in a temporal and sequential mode like the performance of a symphony or the reading of a novel. Just as all artworks are physically embodied, but not identical with the physical body, so too is all software embodied in but not identical with its representation as zeros and ones in physical computer hardware.

A further elaboration of the ontology involves the two terms "prime instance" and "megatype": "... two tokens belong to the same megatype if and only if they approximately share some design from the range of alternative, and even contrary, designs that may be defensibly imputed to each ..." (p. 54). In art, the prime instance would be the original painting or manuscript of a poem or novel. All other tokens of that type would be variations or copies, including non-identical copies, of the megatype of which the original poem is the prime instance. Thus, the manuscript version of a novel by Joyce, for example, would be the prime instance of all the tokens of that type of novel, including the various individual copies printed in various editions of the novel, all of which belong to the megatype of the novel. Other art forms, such as printings of etchings will have no prime instance at all, while an original painting would be a prime instance and all printings or copies of it would be tokens of the megatype. This provides us with a terminology to identify different versions of an artwork as tokens of the same megatype, for instance, the different performances of *Romeo and Juliet*, in different periods, cultures and languages, and even in different media such as the very different film versions of the play by Zephirelli and Lurhmann. In terms of software engineering, this terminology is helpful to describe the different versions of applications (programs) that are continuously released. Each new version of a piece of software, which is released to a customer (for example, a new version of photo-software), would be a token of the megatype "photo-software", whereas each individual use (performance) of this specific copy or instance (probably with its own unique serial-number on the box in which it is sold) would be a token of this specific type.

Mathematical Platonists like Penrose, located algorithms, which are, metaphorically speaking, the "soul" of every computer program, in a Platonic realm of eternal Forms or Ideas (Penrose, 1989). However, this dissertation conjectures that software programs are not located in such a Platonic realm, but rather in Popper's dynamic World 3, the realm of "objective knowledge" which is inhabited by scientific and metaphysical theories. Popper was willing to locate artworks in his World 3, but was equally happy to call the realm

reserved for art, World 4[117]. His point was that theories and artworks are not merely subjective experiences in the minds of their creators and contemplators, but have an objective and non-physical existence independent of individual minds. Unlike Plato's realm of Forms or Ideas, Popper's World 3 is open to change, in the sense of an evolutionary development towards complexity.

In summary, this section on ontology has attempted to illuminate the problem of understanding what software actually *is* by locating software programs in Popper's World 3. In the following chapter, the importance of openness in knowledge processing—which Popper's three worlds metaphysical model highlighted in this section—is extended to a community broader than just the XP team, using Popper's notion of an *open society*. First, however, a critique of the way in which XP is propagated will be provided in the following section using several of the Popperian ideas that have been introduced in this chapter so far.

[117]Popper (1999) (p. 25)

5.4 Popperian Critique of Agile Software Methodologies

This section builds on the criticisms of the XP methodology made by Johnson (2006) [118]. It will show that Popper's principle of falsificationism, which Johnson explicitly uses, is central to his critique even though he does not acknowledge Popper, but rather seems to attribute his approach to Baconianism and verificationism. Popper's ideas are additionally useful for unmasking claims to exclusive knowledge by certain XP leaders, and for understanding the uncritical *adoption* of XP and Agile methodologies by many software engineers. It should be noted that the criticisms that follow relate predominantly to the techniques used by the Agile leaders to *propagate* the Agile methodologies rather than to the stated values, principles and practices underlying their methodologies, which were shown earlier in this chapter to be largely endorsed by Popper's ideas. Therefore, the use of Popper's ideas to criticise this former aspect of the Agile methodologies is not in contradiction to the discussion in the earlier parts of this chapter. Furthermore, Popper's ideas are useful for correcting Johnson's apparent reliance on verificationism, since the latter approach is also vulnerable to the charge of authoritarianism, as argued in section 2.2 of this dissertation on "The History of Scientific Method".

Underlying Johnson's entire criticism in the penultimate part of his book, is the notion of *skepticism*, an attitude which approaches Popper's fallibilist philosophy of *critical rationalism* (as described earlier in this chapter). Firstly, Johnson raises the concern that the "over-enthusiastic and often uncritical adoption of XP and Agile tenets by many in the software development community is worrying" (p. 125) since "it attests to the willingness of many developers to accept claims made on the basis of argument and rhetoric alone" (p. 125) and because it "highlights the professional gulf existing between software engineering and other branches of engineering and science, where claims to discovery or invention must be accompanied by empirical and independently verifiable experiment in order to gain acceptance" (p. 125).

Johnson opens the "Skepticism" part of his book by quoting Francis Bacon: "Argumentation cannot suffice for the discovery of new work, since the subtlety of Nature is greater many times than the subtlety of argument" (p. 125). The fact that Bacon is the only philosopher Johnson mentions in his book may suggest that Johnson was most influenced

[118] Johnson's book, *Hacknot*, contains 46 essays originally published on the Hacknot web site between 2003 and 2006. All citations and hence all page references in this section are taken from this book of Johnson's.

by Bacon's philosophy. However, the main thrust of Johnson's critiques in his book seems to be on the basis of falsification *á la* Popper, rather than on verification *á la* Bacon. Johnson, in his proposed remedy of the uncritical adoption of XP by the XP community, argues for a "Skeptical Software Development Manifesto"[119] whose principles seem to be very Popperian. For instance, the first principle reads "Propositions that are not testable and not falsifiable are not worth much" (p. 126).

Secondly, Johnson criticises two types of propositions in an article entitled "Basic Critical Thinking for Software Developers" (p. 127). The first, *vague propositions*, almost always results in the second, *non-falsifiable propositions*. Johnson argues that the XP community often formulates vague, non-falsifiable propositions like "Pair programming results in better code" which means that such propositions cannot be tested empirically and thereby cannot be determined true or false. Furthermore, such propositions are "beyond the scrutiny of rational thought and examination" (p. 128). As described earlier in this chapter, Popper believes hypotheses should be formulated as clearly and precisely as possible in order to expose them most unambiguously to falsification. Popper argued that non-falsifiable theories belong not to science but to metaphysics (as discussed in the previous section). These ideas of Popper seem to coincide precisely with Johnson's view of non-falsifiable propositions. Furthermore, Johnson argues that before engaging in any debate or investigation, one must ensure that all vague propositions are refined[120]. While it is not certain that this refining process is Popperian in nature (since verificationists also refine their hypotheses), Johnson goes on to refer explicitly to falsification when he states "...refine these propositions into more specific and thereby falsifiable statements" (p. 128). Finally, in the same article, Johnson contends that non-falsifiable propositions "have no place in the software engineering domain because they inhibit the establishment of a shared body of knowledge..." (p. 128).[121]

In another essay within *Hacknot* entitled "Anecdotal Evidence and Other Fairy Tales" (p. 130), Johnson points out that in software engineering, a lot of emphasis is placed on

[119] Johnson parodies several aspects of the XP methodology and usually defines his critical arguments in opposition to an existing 'sacred' or seemingly unquestionable aspect of the Agile approach towards software development. In this case, Johnson defines his "Skeptical Software Development Manifesto" in opposition to the principles of the Agile Software Development Manifesto.

[120] Johnson also specifies some criteria of refining propositions. These criteria seem to coincide with the Karl Popper debate format, originally create by the Open Society Institute, which is widely used in Eastern European and Central Asian high schools (KPD, 2008).

[121] This point can be illuminated by Popper's evolutionary epistemology, namely the growth of knowledge through error elimination and criticism.

a practitioner's experiences since the discipline does not have the benefit of conducting controlled investigation and experimentation. As a consequence, he raises the criticism that the XP community frequently dismisses any comments about the utility of XP or Agile methods unless the person raising the comments has had first hand experience of them. This criticism of Johnson's can be broadened. An important part of Popper's work was to expose appeals to experience as appeals to authority. For example, Popper exposed the empiricists' use of experience as the supposed indubitable foundation of knowledge. What is crucial for Popper is a critical attitude. In Popper's epistemology, experience is not used to justify knowledge claims but to challenge or test them. This critical attitude seems to be lacking amongst certain members of the XP community regarding the propagation and utility of XP.

Finally, Johnson draws an analogy between programming and the scientific method. He selects a method consisting of three phases—*model*, *predict* and *test*—for the analogy. He compares the model phase to a scientific *hypothesis*, the predict phase to a scientific *theory*, and the *test* phase to a scientific *fact*. From the analogy, Johnson identifies what he believes to be a crucial difference between programming and scientific experimentation, namely, that in scientific experimentation, reality is held invariant and we adjust our theory until the theory explains the observed reality and until the two are consistent. In programming, he argues, the model (usually the customer's requirements) is held invariant and we adjust reality (the software) until the two are consistent.[122] However, not everyone would agree with this characterisation of physical science. Kuhn, for example, would argue that reality itself seems to change[123] when there is a paradigm shift and that nature is forced fit the paradigm.[124] The Agile and XP proponents would argue that requirements or "stories" are not invariant and that it is one of the strengths of their methodologies that they can accommodate changing requirements. Nonetheless, one might say that the locus of invariance lie in different places in science and software engineering, even though the invariance might not be absolute in either case.

In summary, it was shown in this section that certain members of the Agile community seem to lack a sufficient awareness of the importance of a fallibilist approach to knowledge,

[122] A similar observation was made earlier in section 5.1, where it was argued that in both science and in software engineering the problem remains invariant.
[123] Kuhn (1970) (p. 111)
[124] Kuhn (1970) (p. 24)

especially in relation to the way in which they *promote* their methodology. However, several of the XP values, principles, and practices, which were shown to accommodate human error quite well, seem to embody an awareness of man's fallibility. By uncovering this tension, a means can now be devised to counter the lack of a fallibilist approach towards the propagation of the XP methodology. The following chapter will investigate this tension in more detail, especially in relation to its implications for the ethos of the XP methodology.

Reflection: Part II

To conclude Part II of this dissertation, which dealt predominantly with the branch of philosophy known as *epistemology*, the following reflection[125] is provided.

The foregoing constitutes an exercise in *applied philosophy*: applied in the sense that, in contrast to *pure philosophy*, there was no discussion, as such, of a general problem (e.g. the problem of knowledge or the problem of science). Instead, the discussion in this part was motivated *occasionally* by the specific problem of *change* related to software engineering methodologies. This part of the dissertation will be incomplete without a (however brief) reflection on the possible limitations of such an approach. Especially in the context of the first chapter in this part of the dissertation, which dealt with Kuhn's ideas, *two questions* can be asked:

(a) Is Kuhn's view of the (history of the) scientific world largely correct, or is his opinion itself merely a "paradigm", in other words, a temporary fashion which is soon likely to be superseded by another one with its own set of believers and dedicated followers?

(b) Regardless of whether or not Kuhn's model is intrinsically plausible, was it applied consistently and appropriately in this chapter to the specific problem in the context of software engineering methodologies?

Question (a) will not be considered further since it is already addressed by the ongoing philosophical discourse with inputs from a great diversity of sources such as, *critical theory* (Frankfurt School), *critical rationalism* (Popper, Albert, et al.), *pragmatism* or *pragmaticism* in their various forms (Peirce, James, et al.), various versions of (meta)-scientific *relativism* (Nietzsche, Feyerabend, et al.), and many others.

[125] Parts of this reflection have been adapted from Northover *et al.* (2007a), a paper which I was the main author of in 2007.

As far as (b) is concerned, it may well be conceded that it is methodologically daring to use (as certain software engineers have done) a model such as Kuhn's, which was conceived as a historic meta-theory about *scientific* theories, to describe aspects of the software engineering discipline, which is, more than anything else, a technical *practice* and **not** a mathematically formulated, predictive scientific "theory", in the classical sense of the term.

However, underlying the practice of every software engineering methodology is undoubtedly some type of "theory", albeit a social theory, which asserts that *if you produce software using this particular methodology, then the outcome is likely to be an effective and efficient software product of the appropriate quality.* This renders the practice of software engineering akin to other practice-oriented disciplines such as medicine or the social sciences, which have to rely for predictive purposes on heuristics, rather than on the precision of mathematics. Therefore, there seems to be an underlying background theory to every software engineering methodology, which is empirically expressed by a multitude of published (yet rarely applied) software metrics, quality criteria, capability maturity models, and the like.

This part of the dissertation investigated whether the use of Kuhn's ideas by certain leading Agile advocates, can constitute such a background theory for Agile software methodologies. It was found that Kuhn's ideas have been used rather superficially in relation to Agile software methodologies, to predict that large-scale change would result if an Agile software methodology is *adopted* to replace completely a more traditional methodology. This prediction assumes that Agile software methodologies can be equated with paradigms, and the shift from traditional to Agile methodologies can be equated with a "paradigm shift"—two topics, which were discussed in chapter 4. With regard to XP in particular, it was shown that this analogy agrees only with the approach to the full-scale adoption of XP in the first edition of *Extreme Programming Explained*, and that it conflicts with the approach in the second edition, namely, the incremental adoption of XP. After applying Kuhn's concepts systematically by analogy to Agile software development in the first chapter of this part, it was also found that Kuhn's theory as a whole, does not account for the daily practice of Agile software development. In particular, from a small-scale (more technical) perspective of software development, a Kuhnian perspective has nothing to say about a computer program designed to calculate, for example, the square of two numbers, that would output the square of 2 to be 5. Therefore, it can be concluded that Kuhn's ideas do not provide a background theory for Agile methodologies, but only a prediction of the

effects of the adoption of XP, which seems to follow plausibly from the approach towards adopting XP that the first edition of *Extreme Programming Explained* advocates.

In the second chapter of this part, it was conjectured that from a *small-scale* perspective of software engineering, where simple issues of *correctness* are at stake, the critical rationalist Popperian ideas of conjecture and refutation, seem to be more appropriate than Kuhn's ideas. It was shown that several XP values, principles and practices are aligned with Popperian ideas. In particular, using Popper's principle of *falsificationism*, which provides a line of demarcation between science and pseudoscience, it was shown, firstly, that software engineering in general, resembles a scientific discipline due to the susceptibility of software to testing. Secondly, it was shown that XP's approach of *incremental design* resembles Popper's model of the *evolutionary theory of knowledge*, despite having to add another phase to Popper's model in order for the analogy to accommodate XP's unique practice of *test-first programming*. At a meta-level, Popper's model was also shown to account for the incremental *adoption* of the XP methodology itself, as advocated by Beck in the second edition of *Extreme Programming Explained*. Finally, Popper's *three worlds metaphysical model*, which emphasises the importance of *objective knowledge* to the advancement of scientific knowledge, was shown to provide a comprehensive framework for understanding how Agile methodologists create and share knowledge, despite their stated reliance on *tacit* or *subjective* knowledge.

In Part III of this dissertation, the topic of discussion will shift from *epistemology* to another well-known branch of philosophy—*ethics*—in order to investigate whether Popper's concept of an *open society* can provide an even deeper philosophical understanding of XP.

Part III

Ethics

The aim of Part III is to investigate whether the *promotion* of XP by certain of its leaders, and the *adoption* of XP by certain members of the software engineering community, is as *open* and *flexible* as the XP values, principles and practices were shown to be in Part II of this dissertation.

Part II transferred concepts from the philosophy of science (in particular, the opposed notions of science of Kuhn and Popper) to software engineering, in order to assess how far Agile software methodologies—and XP in particular—could be considered scientific (and, therefore, rational). On the one hand, it was found that Kuhn's notion of *paradigm shift*, which has been used both explicitly and implicitly—but seemingly problematically—by certain members of the Agile and XP communities to explain or predict a large-scale shift from traditional software methodologies to their own Agile methodologies, was largely inappropriate when applied in detail and in its strict sense to Agile software methodologies. Furthermore, Kuhn's notion of *normal science* was found inadequate to describe the constant state of change of everyday XP software development. On the other hand, it was found that, in terms of small-scale daily software development, XP is closely aligned with Popper's *falsificationism* and *evolutionary epistemology*, and that Popper's *three worlds metaphysical model* contributes to an understanding of the way XP teams create and share knowledge.

In this part of the dissertation, which comprises chapters 6 and 7, concepts will be transferred from *social* engineering to *software* engineering. The common theme throughout both chapters will be the use of Popper's ideas to illuminate the different approaches to *planning* and *managing* software engineering projects.

In the first section of chapter 6, Popper's ideas of *utopian* versus *piecemeal* social engineering, and the *open society*, will be used in an attempt to understand the divergent approaches of traditional versus Agile software methodologists. It will be investigated whether, on the whole, the approach of traditional *Waterfall-like* methodologies resembles utopian (or holistic) social engineering, and whether the approach of Agile methodologies resembles piecemeal social engineering. It will also be investigated, on the one hand, whether many of the difficulties faced by traditional methodologists are partly a result of an assumption of what Popper called an uncritical, or comprehensive, rationalism, typical of authoritarian institutions and societies. On the other hand, it will be conjectured that, in contrast to traditional methodologies, the approach of Agile methodologies—and Beck's XP in particular—seems to presuppose what Popper calls a fallibilism or critical rationalism in

its espoused values, principles and practices. Furthermore, by comparing the XP team to an *open society*, the degree to which the XP practices can be considered democratic, will be assessed, and a criterion will be available for demarcating democratic from authoritarian software team management. Thus, the deeper structure of the argument in this first section of chapter 6 may be understood in terms of differing conceptions of *rationality*, the one more *authoritarian* and the other more *democratic*.

In the second section of chapter 6, the ideas that Kuhn's insulated community of scientists resembles a *closed* society, will be explored. It will be argued that such a closed society feeds into an authoritarian view of science, one which a particular author from the software community, Johnson, has identified as the cultish behaviour of the XP leadership. Johnson's critique seems to be a serious challenge to the avowed openness of the XP methodology, although, it will be argued, his critique seems to relate exclusively to the way XP is *propagated* and *adopted* rather than to the XP values, principles and practices themselves. It will be investigated whether there are parallels between Kuhn's notion of "paradigm shift" and the holistic and irrationalist way in which Johnson claims XP is *promoted*.

In chapter 7, the influence of Christopher Alexander and the authors of *Peopleware*, DeMarco and Lister, on the Agile and XP software methodologies, will be investigated. These specific authors have been chosen for inclusion in this chapter, firstly, because, in their respective disciplines of *architecture* and *software project management*, they argued that sociological aspects are just as important as technological aspects. And from the sociological aspects, often follow ethical implications—the concern of this chapter. Secondly, it is conjectured that there is a similarity between many of the ideas these authors advocate and those which underlie Popper's piecemeal social engineering.

In the first section of chapter 7, it will be conjectured that the use of Popper's piecemeal social engineering to illuminate the Agile approach to software engineering, may have unexpected corroboration in the work of Alexander, an architect whose ideas inspired the *design patterns movement* in software engineering, and continue to have an influence on many of the leaders of the Agile and XP movements, most notably Kent Beck. Of particular focus will be how Alexander's ideas of piecemeal, organic growth in architecture, and his rejection of authoritarian, blueprint engineering, relate to Popper's piecemeal social engineering and to XP's iterative-incremental design approach.

In the second section of chapter 7, the relation between the ideas of DeMarco and Lister

regarding *management style* in the second edition of *Peopleware*, and utopian social engineering, will firstly be investigated. Secondly, their ideas on the office environment, certain of which they explicitly attribute to Alexander, will be discussed. Finally, the authors' ideas on *change* in software engineering will be compared with the respective theories of change of Kuhn and Popper, as well as Beck's ideas of change. Similarly to Alexander, DeMarco and Lister continue to have an influence on Agile and XP methodologies, especially since both are current members of the Cutter Consortium and consult in Agile project management.

The intention of part III is to investigate the issues above and not to provide a comprehensive account of the numerous issues which fall within the scope of computer or software engineering ethics, which are entire disciplines in their own right. The reader is referred to Appendix D where a brief historical account is provided of *Computer Ethics*, and to Appendix E where the definition of the *Software Engineering Code of Ethics and Professional Practice* is provided in its entirety.

Chapter 6

Open and Closed Society Values of Agile Software Development

6.1 Popper's Open Society

Just as Popper provides a criterion for demarcation between *science* and *non-science* in his principle of falsificationism, so too did he provide a criterion of demarcation between *open* and *closed* societies, between authoritarian and free societies. What links both criteria of demarcation is a rejection of authority and an espousal of criticism and fallibility, thus showing the common approach of *critical rationalism* uniting Popper's *epistemology* and his *political philosophy*. According to Magee (1973), Popper's philosophy of science is seamlessly interwoven with his ethical philosophy:

> Rationality, logic and a scientific approach all point to a society which is 'open' and pluralistic, within which incompatible views can be expressed and conflicting aims pursued.

For Popper, the basic question of political philosophy should not be "Who should rule" but "*How can we so organize political institutions that bad or incompetent rulers can be prevented from doing too much damage?*"[1]. According to Popper, this can only be achieved in an open society, with checks and balances to power, and procedures for removing bad leaders without bloodshed:

[1]Popper (1966) (p. 121)

> . . . we may remark the paramount importance in the political arena of the principle of not running the risk of irrevocable and uncontrollable mistakes. It means that democratic political institutions must be concerned primarily with the safeguarding of freedom, especially the freedom to safeguard freedom, and thus with the prevention of irremovable tyranny. (Miller, 1983)

Since Popper was mainly concerned with exposing the enemies of the open society, he did not develop much of a positive account of the open society. Nonetheless, an attempt will be made here to provide a general sketch of his notion of an open society. Perhaps it is best described as a *liberal or social democracy*, since it is based on the individualist ethic of respect for persons, an egalitarian form of justice and the rule of law:

> This individualism, united with altruism, has become the basis of our western civilization. It is the central doctrine of Christianity ('love your neighbour', say the Scriptures, not 'love your tribe'); and it is the core of all the ethical doctrines which have grown from our civilization and stimulated it. It is also, for instance, Kant's central practical doctrine ('always recognize that human individuals are ends, and do not use them as mere means to your ends'). There is no other thought which has been so powerful in the moral development of man.[2]

Magee defines Popper's open society as:

> an association of free individuals respecting each others' rights within the framework of mutual protection supplied by the state, and achieving, through the making of responsible, rational decisions, a growing measure of humane and enlightened life. (Magee, 1973)

Thus, in the open society, a pluralistic society that protects minorities, the state is there to protect the freedoms of the individual. The open society is one without any claims to authority but instead is the mechanism allowing incompetent rulers to be removed from power without violence. Thus, just as Popper rejects the search for foundations of knowledge in *epistemology*, so too does he reject foundations of authority in *politics and society*, and in both encourages criticism[3]. In contrast, in a closed society, the ethic is collectivist and the

[2]Popper (1966) (p. 102)

[3]Although this chapter will refer to open and closed societies, this criterion can be applied equally well to any level of society, to any institution or community within a society, whether a government, corporation, school, university, church, or even a family.

interests of the individual are subordinated to the interests of the collective or the state. Furthermore, in a closed society, criticism and rational debate are discouraged.

Before describing Popper's ideas concerning utopian and piecemeal social engineering and attempting to use them to illuminate the divergent approaches of traditional and Agile software methodologists respectively, an analogy will be drawn between the open society Popper advocates and the XP team Beck advocates. In particular, the XP team itself will be described in terms of an open society, as Popper perceived the community of scientists within the broader open society to be.

6.1.1 The XP Team as an Open Society

While the XP values, principles and practices, which Beck describes in his *Extreme Programming Explained*, are intended to be used specifically by a community of software developers, namely, the XP team, Beck did also consider a broader community than the XP team in his book, as the final chapter called "Community and XP" shows:

> A supportive community is a great asset in software development. This is true whether the community in question is the team itself, like-minded software developers in the local area, or the global community... For open, honest communication to take place in a community, the participants must feel safe and understood... Communities also provide accountability, a place to be held to your word... Communities are a place for questioning and doubt. Each individual's opinion holds value in the community. Conflict and disagreement are the seeds of learning together. Squeltching conflict is a sign of weakness in a community. Valuable ideas can withstand scrutiny. Members of a community don't commit to full-time unanimity; they agree to respect each other while they work out their disagreements. Compliance is not a requirement for participation in a safe community... As a community, we can accomplish more that we ever could in isolation. (pp. 157-158)

This quotation shows convincingly that XP, as advocated by Beck, shares fundamental values in common with Popper's notion of an open society. For instance, the broad values of an open society are evident in "open, honest communication", "accountability" and "respect", the values of "egalitarianism" and individualism are evident in "Each individual's opinion

holds value...", and the Popperian critical attitude is evident in "questioning and doubt", "Conflict and disagreement..." and in "...ideas can withstand scrutiny". Furthermore, it is clear from "Squeltching conflict..." and "Compliance is not a requirement..." that Beck, like Popper, would reject properties of a closed society.[4]

While a comparison of the *XP team to a scientific community in the broader open society* would be equally important to a comparison of the *XP team to an open society as a whole*, this section will only provide the latter comparison, since the former was addressed to a large extent already in section 5.1.

Important for an understanding of the comparison between the values of an XP team and those of an open society in the following section, is the following: it may seem that by emphasizing the importance of the team over the individual members, Beck contradicts Popper's belief in the supremacy of the individual. However, Popper believes that "The objectivity of science ... [is] in the hands not of individual scientists alone, but of scientists in a scientific community" (Miller, 1983). Therefore, if the XP team members are compared to scientists and the XP team itself is compared to a scientific community, Beck's view does not seem to contradict Popper's. The following section will elaborate on this analogy by investigating which of the XP values, principles and practices, if any, are aligned with the values of an open society, as advocated by Popper.

The Open Society and the XP Values, Principles and Practices

Firstly, it can be argued that the two related XP **values**[5] of *courage* and *respect* are essential in an open society. In XP, *respect* is said to underscore *courage*: a team member will only contribute to a project in the knowledge that the contribution will be valued and respected

[4]In my experience, despite the appeal of Beck's view of software development, the approach he describes is not always followed in practice. In the large multi-national organisation I worked for, the hierarchical structure of the organisation meant that it tended towards bureaucracy and authoritarianism, thereby discouraging criticism and open debate. Such organisations are similar to those that Beck describes as dismissing criticism and avoiding conflict. However, I have recently heard accounts of a trend emerging, where there is a shift to involve people from all levels of the organisation in initiatives, and where each individual's performance is measured according to all of the people that individual provides service to, including the individual's superiors, people who work for the individual, and people who work with the individual. This trend seems to be quite encouraging. Beck's view seems to be important, since it does not describe a utopia where software developers are imagined to work in harmony and uniformity, but a realistic software community that recognises plurality and individuality, with all the differences of opinion such a community gives rise to.

[5]Note that the XP values, principles and practices, which will be discussed in this section, are categorised as such by Beck in his *Extreme Programming Explained*. The categorisation can, in many cases, be contested. For instance, it can be contested whether *feedback* is a value, and whether *diversity* is a principle. Nonetheless, Beck's categorisation is used here to be consistent with his *Extreme Programming Explained*.

by other team members. In turn, each team member takes the responsibility to listen and respond to the proposals of others. Similarly, in an open society, courage is an essential basis for critical debate. With regard to *respect*, Beck argues that respect underlies each of the other four XP values: *communication*, *simplicity*, *feedback* and *courage*. He states "If members of a team don't care about each other and what they are doing, XP won't work. If members of a team don't care about a project, nothing can save it"[6]. Furthermore, Beck states: "Every person whose life is touched by software development has equal value as a human being. No one is intrinsically worth more than anyone else. For software development to simultaneously improve in humanity and productivity, the contributions of each person on the team need to be respected"[7]. Popper's view of *respect* is similar to Beck's since he espouses the Kantian principle of *respect for persons*. In a democratic open society, each individual has equal value as a human being and, therefore, each individual's opinions should be respected. In a similar way that many of XP's practices, as advocated by Beck, create opportunities for critical and open discussion, so too does Popper argue that the state should actively nurture critical discussion.[8]

The XP **principles** of *humanity*, *diversity*, and *accepted responsibility* also seem to coincide with the values of an open society. Regarding *humanity*, Beck argues:

> People develop software... Acting like software isn't written by people exacts a high cost on participants, their humanity ground away by an inhumane process that doesn't acknowledge their needs.

Similarly, in an open society, Popper promotes a humanitarianism based on individualism, where the interests of the individual are not subordinated to the interests of the collective. Popper shows how individualism is often confused with egoism, and altruism with collectivism. For Popper, the true Christian ethic of western civilisation is one of altruistic individualism.[9] In software development, one finds both "open" teams, where team members are respected as individuals and trusted with responsibility, and "closed" teams, where team members are dictated to by an authoritarian leader and not trusted with responsibility.

Regarding the XP principle of *diversity*, Beck argues that XP teams need diversity "to bring together a variety of skills, attitudes, and perspectives to see problems and pitfalls,

[6]Beck & Andres (2005) (p. 21)
[7]Beck & Andres (2005) (p. 21)
[8]Fuller (2003) (p. 107)
[9]Popper (1966) (p. 100)

to think of multiple ways to solve problems, and to implement solutions" (p. 29). He also points out that along with diversity comes *conflict*, in the sense of competing ideas. Similarly, in an open society, the contributions from its diverse citizens (who often offer competing proposals) are relied upon for a plurality of views.[10]

Regarding the XP principle of *responsibility*, members of an XP team, according to Beck, are not assigned responsibility by an authority but instead accept responsibility[11]. This is aligned with Popper's contention that responsibility thrives in the individualist ethos of an open society, whereas the authoritarian ethos of a closed society tends to discourage individual responsibility and questioning, and to encourage obedience to authority instead.

The XP **practices** of *whole team*, *sit together*, *real customer involvement* and *informative workspace* also have parallels in the open society. The first three practices are interrelated and all facilitate the free exchange of ideas. According to Beck, it is important for XP team members to be in close physical proximity since this enhances open, face-to-face communication.

Sit together predicts that the more time spent in close physical proximity, the more humane and productive the project will be. This practice, therefore, reinforces the XP principle of *humanity*, linking it to Popper's humanitarianism.

Whole team suggests that a variety of people work together in inter-related ways to make a project more effective, thereby encouraging a sense of community among its members by fostering acceptance and belonging. More than the others, this practice seems to be contrary to Popper's ideas, since he accepted the necessity in the open society of an impersonal and abstract social life. However, Beck clearly states that "What constitutes a 'whole team' is dynamic". By this, he means that if the team lacks a certain skill or attitude, a person with this skill should be brought onto the team. Similarly, if a certain skill is no longer needed on the team, the person with this skill can move to another project team[12]. Therefore, the sense of community which XP teams enjoy, is in relation to a continuously changing

[10] Of course, societies do not have an ultimate goal to produce a coherent product, unlike a software development team. Therefore, perhaps a society can embrace different points of view more readily than a software development team can. In this regard, it is interesting to note that Brooks' notion of *conceptual integrity* (Brooks, 1975) (ch. 4) has been known to produce a wider range of solutions.

[11] "Accepted responsibility" is evident in XP when, for instance, the same developer who volunteers for a particular task is encouraged to estimate its cost and time as well, thereby committing to be able to complete the task within the estimated time and budget. The resulting sense of ownership of the task will most likely encourage accountability too, another essential quality of scientists in an open society.

[12] Note that this suggestion of Beck's cannot be interpreted as treating individual team members *merely* as means to the team's ends since the individual receives compensation in the form of a wage for providing these skills. In other words, there is reciprocity between the team and the individual.

open community, due to the dynamic nature of team formation. Another aspect of the XP team (which comprises testers, interaction designers, architects, project and product managers, executives, technical writers, users, programmers and human resources)—its apparent holism— may also seem contrary to Popper's ideas, since he rejected holism. However, this is not the case since Beck clearly supports individualism. Each team member is considered an authority in his/her specific field of expertise. For example, the project manager has similar responsibilities to the team members as the state has to its citizens in an open society. In relation to programmers, project managers should act as facilitators rather than authoritarian leaders, and they should ensure that conditions are conducive to productive software development. They should give developers the responsibility to make all the technical decisions, trusting them to get done the work they've chosen and estimated. XP recognises that individuals are the most important factor of success in a project, and the successful functioning of the XP team largely depends upon the individuals.[13]

It can be argued that the emphasis XP places on users being a core part of the XP team—as indicated by the practice *real customer involvement*—encourages the formation of a *broader* open society. XP developers, like scientists in an open society, are not insulated from the broader, lay community, and the user provides considerable critical input.

The final XP practice mentioned above, that of *informative workspace*, (which was discussed earlier in sections 2.4 and 5.3), facilitates transparency and openness not just within the XP team but also more broadly. Beck argues "An interested observer should be able to walk into the team space and get a general idea of how the project is going within fifteen seconds" (p. 40), clearly pointing to the openness of the development style. Not only is the workspace informative because it contains story cards and visible charts on walls, but also because of the way the workspace is arranged. It creates space for critical discussion, which also supports error elimination. Furthermore, certain XP practices, like stand-up meetings, encourage individuals to openly express and defend their ideas in a democratic setting. Regarding the XP workspace, Beck says:

> We have nothing to hide. That's the plan, open and accessible, that reflects the kind of relationships that make for the most valuable software development.

Similarly, in an open society, of central importance is creating and maintaining conditions

[13]Social dynamics and psychological tendencies, therefore, are of central importance to XP software development.

in which open questioning is possible and where criticism is actively encouraged.

In the following section, Popper's ideas regarding utopian social engineering will be compared to the traditional approach towards software development, in an attempt to investigate whether, on the whole, the approach of the traditional software developer resembles the approach of the utopian (or holistic) social engineer. XP's critique of many of the traditional software methodologist's beliefs and practices will also be pointed out.

6.1.2 Utopian Social Engineering

In this section, a comparison will be made between the practices of a typical utopian social engineer and those of a traditional[14] software engineer. This comparison will attempt to illuminate the reason why so many software projects fail. In particular, it will be conjectured that the different approaches towards software development of traditional and Agile methodologists are based on different ideas of rationality. It will be investigated whether the high rate of failure of traditional methodologies may at least partly be explained by their adherence to what Popper believes is a faulty model of rationality, a model based on the principle of sufficient reason, which Popper calls *comprehensive rationalism*. Popper bases his own type of piecemeal social engineering, which will be discussed in the following section, on *critical rationalism*.

Utopian social engineering, according to Popper, aims at remodeling the "whole of society" according to an idealistic and unchanging plan or *blueprint*. Utopian societies are characteristically controlled by a few powerful individuals who believe that social experiments are only of value when carried out holistically. The utopian approach may *seem* to be scientific, based as it is on the following reasoning:

> *Any action must have a particular aim. The action is considered rational if it seeks its aim consciously and determines its means according to this aim. In order to act rationally, therefore, the first thing to do is to choose the aim. There must be a clear distinction between the intermediate aims and the ultimate aim.*[15]

However, despite its apparently rational reasoning, several of the premises of the utopian approach can be questioned, since:[16]

- So much of society is reconstructed at once under utopian social engineering that it is unclear which measures are responsible for which results. And if the goal is in the distant future, it is difficult to say if the results are in the direction of the goal or not.

[14] The word "traditional" is used throughout this section (and, indeed, throughout this dissertation) to represent approaches towards software development which are based on *Waterfall-like* methodologies. Stereotypically, these plan-driven methodologies are less flexible and less able to cope with change than the more recent Agile methodologies. Instead, their emphasis is on predictability, repeatability and optimisation as shown in section 2.3.

[15] Paraphrased from Popper (1966) (p. 161).

[16] The following critique is adapted from Popper (1966).

- The complete reconstruction of society takes a long time and is, therefore, unlikely to be achieved during the lifetime of a single person. It can be imagined that not all utopian social engineers who work on the reconstruction of society over the decades will necessarily pursue the same ideal. It is even possible that the objectives—according to the social engineer who conceived the original blueprint—may change.

- Social life is too complicated to predict, therefore, unintended consequences are inevitable, resulting in unanticipated events. The greater the holistic changes, the greater the unintended consequences.[17]

- Not everyone will agree about the ideal society so the utopian social engineer will be inclined to impose his social principles on society and discourage criticism, which would lead to authoritarianism. By forbidding the critical examination of his blueprint, his inevitable mistakes will also be discovered much later than necessary.

- Since scientists cannot predict everything, the idea of a totally planned society is untenable.[18]

- The goal of the utopian social engineer to redesign society completely from scratch is untenable because he, along with his blueprint for design, is part of the system that he plans to redesign.

- The notion of an unchanging plan or blueprint is untenable because change is continuous.

- The ultimate end is not an end in fact because history is continuous.

- Ideal societies are unattainable by definition because they are ideal.

Although Kuhn did not extend his philosophy to a broader community than the scientific community, there seem to be significant similarities between a blueprint of society and a Kuhnian paradigm. Indeed, as section 2.2 showed, Fuller links the two: "Most actual science—what Kuhn calls 'normal science'—consists of little more than the technical work of fleshing out the paradigm's blueprint."[19]

[17] Ironically, the holistic engineer would most likely be forced into piecemeal improvisation as a result.
[18] Popper criticised historicist theories by reasoning that if one could predict future knowledge, one would have it already and it would not be the future. In other words, future discoveries would be present discoveries.
[19] Fuller (2003) (p. 19)

By developing a blueprint for redesigning society before considering ways for its realisation, and by assuming that societies can be designed according to this unchanging blueprint, the utopian social engineer, according to Popper, incorporates the same mistaken notion of *certainty* as the traditional scientist following the inductive method of science. The appeal of utopian societies can perhaps be explained by the surety they provide, namely, that individual citizens will not be burdened with the responsibility, which comes with freedom, and that they will not be faced with difficult choices or decisions, nor their consequences. In other words, citizens would be comfortable with the security that such a pre-critical[20] society provides, and they would be content to transfer the responsibility of decisions to the utopian social engineer who supposedly has their best interests at heart. These conditions, according to Popper, are dangerous since the power that is bestowed on the utopian social engineer often leads to corruption. In this regard, Popper cites Lord Acton's dictum "All power corrupts, and absolute power corrupts absolutely"[21].

Finally, the utopian approach, according to Popper, is also connected to aestheticism[22]: the utopian social engineer aims to free society from ugliness by reshaping it according to an ideal of beauty[23].

The remainder of this section will investigate the degree to which the traditional approach towards software engineering resembles the utopian approach towards social engineering.

The (often mistaken) assumption made by many traditional software methodologists is that project requirements are fixed before the project starts. This is a fondly held assumption because fixed requirements allow a project to be planned with more certainty[24]. Consequently, in a similar way to the utopian social engineer who advocates blueprint social planning, traditional methodologists advocate comprehensive a priori design as a prerequisite to software development. Once the software design blueprint is in place, changes to it are kept to a minimum throughout the project life cycle, and any changes to the requirements are often discouraged. Much like the utopian social engineer, the traditional

[20]The degree to which criticism is accepted by a society is a fundamental distinguishing characteristic between open and closed societies. This will become evident once piecemeal social engineering is discussed in the following section.

[21]Popper (1966) (p. 137)

[22]Popper (1966) (p. 165)

[23]An contemporary example of this is the Bolsheviks' attempt to reshape Russia as a communist utopia. This ideal is also embodied in Plato's vision of the *Form of the Good*, which he developed in *The Republic*.

[24]Note that certainty has nothing to do with objective truth, but is merely a subjective psychological state.

software methodologist tends to believe in the infallibility of his approach which, if followed strictly, will lead to a greater chance of success. The approach of the traditional software methodologist is usually to design software systems completely from scratch. Inevitably, the traditional software methodologist imposes his idea of the software process on others much like the utopian social engineer imposes his plans on society. The inherent rigidity and authoritarianism of this approach, reminiscent of *Taylorism*[25], is clearly evident from this description. In addition, the tendency towards inflexibility and software bureaucracy, tends to stifle both creativity and innovation, two qualities which Popper believes are essential to a critical rationalist tradition of science, and which are often invaluable to software engineering.

The remainder of this section will investigate the extent to which the XP methodology, in particular, rejects the equivalent of a utopian approach to software engineering.

Firstly, while arguing for an iterative approach towards software development, Beck & Andres (2005) implicitly criticise the premises of what can arguably be called a utopian approach to software engineering:

- We may not know how to do it "right". If we are solving a novel problem there may be several solutions that might work or there may be no clear solution at all.
- What's right for today may be wrong for tomorrow. Changes outside our control or ability to predict can easily invalidate yesterday's decisions.
- Doing everything "right" today might take so long that changing circumstances tomorrow invalidate today's solution before it is even finished.[26]

All three points highlight why XP rejects blueprint planning. The first points out that often there may be more than one solution to a problem, which implies the need for more than one design. However, the definitive blueprint design of utopian planning can only accommodate a single design alternative. The second point highlights the volatility of software development, showing that the utopian engineer's belief in being able to predict everything

[25]The word *Taylorism* derives from the name of the industrial engineer who sought to improve factory productivity and efficiency by applying the methods of science. According to Beck, symptoms of Taylorism in software development are rife. Cases abound, for example, where a person in authority estimates the work that a subordinate has to carry out without the subordinate's input, or where an "élite" group within the organisation (such as a design or architecture group) prescribes exactly how others who are not part of the group should do their work.

[26]Beck & Andres (2005) (p. 19)

and capture it in an unchanging blueprint, is unrealistic. The third point highlights that attempting to perfect the blueprint may take so long that the objectives may change in the meantime.

Instead of blueprint planning, XP's approach is, according to Beck: "Being satisfied with improvement rather than expecting instant perfection, we use feedback to get closer and closer to our goals". The similarity between Beck's view and Popper's in terms of planning is striking, and, once again, questions Beck's use of Kuhn's concept of a "paradigm shift" since a Kuhnian paradigm is just such a blueprint for scientific research, as Fuller's quote above showed. Beck also believes that, no matter what the circumstances, individuals and teams can always improve. Therefore, unlike the utopian engineer, Beck does not advocate striving for perfection. Furthermore, Beck encourages software engineers to abandon the surety that "I know better than everyone else and all I need is to be left alone to be the greatest". On the other hand, however, Beck also states that "XP is a path of improvement to excellence. . . ", which suggests a utopian ideal. Nonetheless, he argues that XP is not to be interpreted as "a set of rules to follow that guarantee success". In other words, the XP values, principles and practices themselves should not, according to Beck, be considered a blueprint providing assurance that, if followed strictly, will certainly lead to successful software projects. Similarly, Popper's principles of *falsificationism* and *error elimination* are anti-utopian.

With further regard to the *volatility* in the software industry, Beck implicitly points out the weaknesses of an unchanging blueprint when he states:

> No fixed direction remains valid for long; whether we are talking about the details of software development, the requirements of a system, or the architecture of a system. Directions set in advance of experience have an especially short half-life. Change is inevitable, but change creates the need for feedback.[27]

The points Beck makes here are similar to several points Popper raises in his critique of utopian social engineering. Both Beck and Popper believe change to be continuous and inevitable. They also both believe that any fixed plan will be invalidated if it is not open to feedback and change. XP's view of a plan, according to Beck, is "A plan in XP is an example of what *could* happen, not a prediction of what *will* happen". This view is clearly

[27] Beck & Andres (2005) (p. 19)

anti-utopian and also demonstrates the conjectural nature of XP's approach. Instead of a master plan, Beck suggests a *shared* plan:

> A mutually agreed upon plan, adjusted when necessary to reflect changing reality, suggests a respectful, mutually valuable relationship... The plan can change to fit the facts as they emerge... Plans are not predictions of the future. At best, they express everything you know today about what might happen tomorrow. Their uncertainty doesn't negate their value.

The shared plan in XP also considers the people who are affected by the plan just as the piecemeal social engineer's plan considers the citizens in the part of society it will affect. Furthermore, Beck's belief that "Plans are not predictions of the future" resembles one of the underlying beliefs of Popper's criticism of a type of philosophy called *historicism*.

XP's view of *authority* is also contrary to that of the utopian approach, in which the utopian social engineer enjoys absolute power. According to Beck:

> The principle of the alignment of authority and responsibility suggests that it is a bad idea to give one person the power to make decisions that others have to follow without having to personally live with the consequences.

Beck's view is echoed by Popper's criticism of handing all power to a philosopher-ruler who has received extensive training in how to rule.

Finally, with regard to XP's approach towards *change*[28], Beck warns against making large-scale changes, advocating instead *baby steps*:

> It's always tempting to make big changes in big steps... Momentous change taken all at once is dangerous. It is people who are being asked to change. Change is unsettling. People only change so fast... Under the right conditions, people and teams can take many small steps so rapidly that they appear to be leaping... Baby steps acknowledge that the overhead of small steps is much less than when a team wastefully recoils from aborted big changes.

In conclusion then, the comparison between utopian social engineering and traditional software development, as well as XP's criticism of many practices of a utopian approach to

[28] Beck uses the analogy of "learning to drive" (Beck & Andres, 2005) (11) when describing XP's approach towards change. Essentially, if one's goal is to reach another part of town, after setting out in the general direction, one would need to make continual adjustments to keep on the road, pass other vehicles on the road, stop at traffic lights, and so on. One would not dare to set out and refuse to make any adjustments.

software development, have provided some insight into the failure rate of traditional methodology projects: it cannot be assumed software developed using a blueprint will necessarily result in a good or even workable software system; more likely, continual changes will be required along the way, which will necessitate an incremental approach of making small adjustments. Popper's ideas were shown to help expose the faulty notion of comprehensive rationalism behind any type of holistic engineering, and Beck's criticisms of traditional methodologies were shown to be compatible with Popper's critique of utopian social engineering.

In the following section, it will be investigated whether XP's approach of making small changes to the software, and releasing these small changes incrementally to their customers, resembles Popper's piecemeal approach to social engineering, especially the manner in which the piecemeal social engineer conducts her work.

6.1.3 Piecemeal Social Engineering

This section will investigate whether the approach of the XP software developer—as characterised by Beck—resembles the approach of the piecemeal social engineer—as characterised by Popper—and will conjecture that the two approaches share in common a fallibilist idea of rationality. Any counter-evidence for this conjecture, will also be taken into account.

Piecemeal social engineering, in contrast to utopian social engineering, does not aim to redesign society "as a whole", according to Popper. Instead, it attempts to achieve its aim in a humane and rational manner, by small adjustments and readjustments, which can be continually improved upon. It is a cautious, critical approach that acknowledges the uncertainty introduced by the "human factor". At this point, it is important to remember the discussion in section 5.1, which stated that Popper *prescribes* revolutionary thinking in science but *proscribes* revolutionary activity in society. Whereas in the former approach, Popper encourages scientists to propose bold and daring theories (which, having withstood the harshest criticism, would have widespread consequences if adopted), in the latter approach he warns social engineers to be modest and cautious in making changes to society, and to make these changes in small steps. The latter approach is deliberately much less risky since its aim is to minimise possible harmful effects that social engineering could have on the lives of human beings. An example of a piecemeal social engineering experiment may be the introduction of a new pension fund scheme into part of society. A piecemeal approach in this case may involve introducing the scheme only to new employees in a company while the existing employees remain on the old scheme. If, over a few years, the new scheme outperforms the old, the piecemeal social engineer may decide to introduce the scheme more broadly. It is important to note that the piecemeal approach does not necessarily imply a small-scale impact on society, although it will most certainly have a smaller impact than the utopian engineer's complete redesign of society. In our example, the new pension fund scheme will have a large impact on certain employees but no impact at all on others. For Popper, according to Magee, the main premise of the piecemeal approach is:

> We do not know how to make people happy, but we do know ways of lessening their unhappiness.[29]

Consequently, the task of the piecemeal social engineer is to "adopt the method of searching

[29] Magee (1973)

for, and fighting against, the greatest and most urgent evils of society, rather than searching for, and fighting for, its greatest ultimate good[30]. This approach is analogous with Popper's principle of falsificationism in science. It draws attention to solving social problems instead of the pursuit of an ideal or utopian society. Piecemeal social engineers make their way step-by-step, consciously and actively searching for mistakes in their method. They are aware that they can learn from their mistakes and know that if anything goes wrong, their plan can be stopped or even reversed if necessary. The damage would not be too great and the readjustment not too difficult.

The most striking similarity between Agile software development and piecemeal social engineering, as advocated by Popper, is their common rejection of an overall blueprint for design. Agile methodologists, in contrast to traditional methodologists, argue that software projects are usually too complex, and requirements too volatile, to guarantee the correctness of comprehensive a priori software design, since many project activities are progressive and uncertain. Instead, they believe that software designs are usually the result of slowly acquired insights rather than of knowing (or thinking they know) everything at the start of the project. Specifications of software functionality, usability, and structure, for example, cannot be fully known before software is designed and implemented. Therefore, Agile methodologists place less emphasis on comprehensive a priori design than traditional methodologists, and more emphasis on practices like *incremental design*, which enable them to adapt to changing circumstances. According to Beck: "The better the team is at incremental design, the fewer design decisions it has to make up front".[31] Agile methodologies can, therefore, be categorised as adaptive rather than anticipatory. They advocate that software be built and released in an incremental manner, through a process of short iterations with continuous feedback. During this process, software is built in a "bottom-up" fashion, in small steps as opposed to large leaps. Feedback ensures that the development process remains flexible, which means that change can be accommodated more easily and errors can be eliminated as quickly as possible. Due to their flexibility, Agile methodologies, unlike traditional methodologies, *embrace change* rather than avoid it.

Agile software development and piecemeal social engineering also share a common focus

[30]Popper (1966) (p. 158)

[31]Some may argue that the approach of incremental design is only tenable if software engineers are expert designers to start with, and have a good sense of what functionality can be safely left out, as well as a plan of how to introduce the functionality later, if necessary. Certain aspects—such as, security, parallel access, and error handling—can be very difficult to introduce afterwards.

on the *present* rather than the future. XP specifically emphasises the present through the core value of *simplicity*:

> To make a system simple enough to gracefully solve only today's problem is hard work... [One can] achieve simplicity by eliminating unneeded or deferrable requirements from today's concerns. (pp. 18-19)

To achieve simplicity, the XP software developer assumes a position of deliberate shortsightedness and minimalistically implements the highest priority stories, which are known today to be required. This approach ensures that the economic value of the project is maximised. Any unneeded or deferrable requirements are eliminated from present considerations. In other words, the XP software developer will not implement functionality for potential future requirements because of the inherent uncertainty of the future. This focus on the immediate problem is similar to the piecemeal social engineer's focus on identifying the most urgent problems in society and bringing about a rapid solution to them. Following a similar approach to the social engineer, the XP software developer chooses the smallest release that makes the most business sense "so there is less to go wrong before deploying and the value of the software is greatest". The observation that "less can go wrong" is precisely the same reasoning that encourages a social engineer to adopt a piecemeal approach rather than a utopian approach towards social engineering.

Another of XP's practices, *incremental deployment*, shares fundamental underlying values in common with piecemeal social engineering. Just as the piecemeal social engineer is modest and cautious in making changes to society, so too is the Agile software developer modest in deploying changes to the user community. *Incremental design* leads naturally to *incremental deployment*, whereby small changes to the software are released to the user community at a time. The impact on the community is, therefore, minimised. This approach is in contrast to the utopian approach in which large-scale deployments are made, thereby affecting a large part of the community with little regard for the impact on them. According to Beck: "Big deployments have a high risk and high human and economic costs". XP's approach of incremental deployment is underpinned by the principle of *flow* which suggests that "more value is created in a smooth, steady stream of software than in occasional large deployments". For instance, when replacing a legacy software system, XP teams will not deploy the entire new system in one release at the end of the project. Instead, they will

deploy small pieces of functionality throughout the project's life cycle, running the new system and the legacy system in parallel for some time. This approach is arguably more humane since it considers the impact on the end users.[32] Instead of discarding the legacy system abruptly and finally, which would require the users to abandon their old way of working completely, the users are given the opportunity to learn the new system in stages as it is deployed. The importance of this approach is evident when considering the findings of the authors of *Peopleware* (DeMarco & Lister, 1999). These authors argue, as will be shown in section 7.2, that people fiercely resist change of whatever kind. Therefore, if XP's incremental deployment practice reduces the extent of change to the software, this will, in turn, require less change on behalf of the users. This approach is aligned with that of the piecemeal social engineer, whose aim it is to minimise the impact of change on society as a whole.

Having shown that Popper's notions of an *open society* and of *piecemeal social engineering* are aligned with many of the values of the XP team, and of the XP software development approach, the following section aims to investigate the reasons why certain leading figures of Agile software methodologies have, either implicitly or explicitly, used Kuhnian concepts to explain aspects of their methodologies. In particular, it will be investigated whether there are parallels between Kuhn's notion of "paradigm shift" and the holistic and irrationalist way in which—according to Johnson's critique of XP in his book *Hacknot* (Johnson, 2006)—the XP methodology is promoted by its leaders.

[32]Some may argue that an approach of running the new system and the legacy system in parallel, would affect end users negatively too. Two systems running in parallel would require duplicate entry of data (and, therefore, decreased productivity), and double the costs to run two sets of infrastructure, two sets of software licenses, and so on.

6.2 Kuhn's Closed Community of Scientists: Implications for XP

The previous chapter revealed the shortcomings of using Kuhn's ideas to account for the day-to-day practice of Agile software development. Nonetheless, Kuhnian ideas have been used by several leading figures of the Agile and XP communities in relation to their methodologies. The aim of this section is to investigate the extent of the influence of Kuhnian ideas on the ethos of the XP methodology.

Since Kuhn's ideas enjoy a widespread hegemony, which is not always questioned, an overview of Steve Fuller's critical account of Kuhn's philosophy will first be provided here[33]. Then, it will be investigated whether this critical account is corroborated by Johnson (2006), whose critique of the XP methodology in his book called *Hacknot*, was already partly investigated in section 5.4. Whereas in that section, it was shown that Johnson uses Popperian concepts such as *falsificationism* to criticise the XP methodology, in this section it will be investigated whether certain aspects of XP, which Johnson's criticises (and which he does not relate to any specific philosopher or philosophy), have parallels in Kuhn's notion of a "paradigm". The discussion in section 5.4 also concerned *epistemology*[34] specifically, whereas the discussion in this section relates more closely to *ethics*.

It is conjectured that Johnson's criticisms are so incisive because he criticises the way XP is *propagated*, revealing certain authoritarian tendencies. It will be investigated whether these tendencies have parallels in Popper and Fuller's criticisms of Kuhn's philosophy, which may show the applicability of Kuhnian concepts to understanding the conduct of the XP leadership, and the promotion of the XP methodology. On the other hand, Johnson does **not** criticise the declared values, principles and practices of XP, perhaps because these embody attributes contrary to the closed community of a cult that Johnson finds so objectionable in the way XP is *propagated*.

[33]Note that Fuller's views of Kuhn are not mainstream. In fact, his approach is revisionist in relation to the *Kuhn vs Popper* debate. Fuller is a sociologist more so than a philosopher. Therefore, he considers not only the argument in itself, but also its context and the motives behind it.

[34]In particular, a comparison was made between a critical scientific method and the seemingly uncritical approach of certain of the XP leaders.

6.2.1 Fuller's Critical Account of Kuhn's Philosophy

Recasting Johnson's criticisms in Kuhnian and Popperian terms—especially his argument that the XP community, in particular its leaders, behave like the priests of a cult—one finds unexpected corroboration in Fuller's statement that "the deep structure of the Kuhn-Popper debate [is] religion" since "both science and religion are preoccupied with justifying beliefs"[35]. Fuller later quotes John Watkins on Kuhn's religious attitude to science:

> My suggestion is, then, that Kuhn sees the scientific community on the analogy of a religious community and sees science as the scientist's religion. If that is so, one can perhaps see why he elevates Normal Science above Extraordinary Science; for Extraordinary Science corresponds, on the religious side, to a period of crisis and schism, confusion and despair, to a spiritual catastrophe. (p. 150)

Indeed, Kuhn himself writes about "... the conversion that we have been calling a paradigm shift" (p. 150) and:

> ... scientific communities have again and again been converted to new paradigms. Furthermore, these conversions occur not despite the fact that scientists are human but because they are. Though some scientists, particularly the older and more experienced ones, may resist indefinitely, most of them can be reached in one way or another. Conversions will occur a few at a time until, after the last hold-outs have died, the whole profession will be practicing under a single, but now a different, paradigm. We must therefore ask how conversion is induced and how resisted. (p. 152)

Kuhn also uses other terms more appropriate to a religious context than a scientific one, namely, "faith":

> The man who embraces a new paradigm at an early stage must often do so in defiance of the evidence provided by problem-solving. He must, that is, have faith that the new paradigm will succeed with the many large problems that confront it, knowing only that the older paradigm has failed with a few. A decision of that kind can only be made on faith. (p. 158)

[35] Fuller (2003) (p. 150)

Fuller further elaborates on Watkins' insights, pursuing the analogy between Kuhnian scientists[36] and monks[37]:

> It turns out that religious conversion is Kuhn's model for the paradigm switch that occurs during a scientific revolution. Moreover, the analogy between a scientific discipline and a religious order had already been developed as a self-styled 'post-critical' philosophy by one of Kuhn's influences, the Catholic chemist Michael Polanyi (pp. 1891-1976). Polanyi[38] was largely responsible for shifting the Anglophone public image of scientists in the 1950s from heroic individuals like Galileo and Darwin who tried to change the world to a more anonymous community of specialists focused on cultivating their craft. This monkish view of scientists, which Kuhn popularised in the following decade as 'normal science', was designed to protect the autonomy of science from policy-makers in both the Capitalist West and the Communist East who, during the Cold War, were keen on converting science into means for larger political ends. The telltale sign of this monastic turn in Polanyi and Kuhn is that the entire sociology of science is reduced to the process of training initiates for a life of total commitment to their paradigm, by virtue of which their judgment will be largely unquestioned in the larger society and questioned only on technical matters within their own community. (p. 102)

The irony of the scientific community's attempt to isolate itself, in Kuhnian style, from politics, was that they were thereby more easily used for political ends. Kuhn avoided "acknowledging the technological dimension of modern science" (p. 89) so as to keep science insulated from real world situations and policy decisions and avoided "asking whether the instruments used in experiments were inspired and/or applied in a military-industrial setting outside the experimental context" (p. 89).

According to Fuller, this resulted in science being sullied in the World Wars (and later the Cold War) as a result of which Kuhn (and the logical positivists) developed a reaction

[36]The term "Kuhnian scientists" should be understood by the reader to refer to scientists who have adopted a scientific paradigm, and whose activities are described by Kuhn's philosophy.

[37]It is interesting to note, as will be discussed in Appendix F, that the Finnish philosopher, Himanen, also contrasts the open Academy (where, according to him, knowledge is shared freely) with the closed monastery (where knowledge is jealously guarded) when he discusses the open source software community.

[38]The discussion of Polanyi's ideas in section 5.3 should be bourne in mind here, and the link between Kuhn and Polanyi noted.

formation to protect themselves from the sordid truth:

> In response, they promoted excessively idealised visions of science that were the opposite of the tendencies they rejected in the science of their day. For Kuhn, the ultimate corruption of spirit by matter—or science by technology—had been epitomised by the involvement of the founders of modern particle physics in the US and German atomic bomb projects. The theorists of the ultimate constituents of matter and energy were also the designers of the ultimate weapons of mass destruction. (pp. 89-90)

Having summarised the relevant parts of Fuller's critique of Kuhn above, the following section will proceed to discuss Johnson's critique of the XP community, and will investigate whether certain aspects of XP that Johnson criticises have parallels in Fuller's critique of Kuhn's ideas.

6.2.2 Johnson's Critique of XP

Firstly, in a chapter of *Hacknot* called *Extreme Deprogramming*, Johnson argues that XP, with its groups of ardent supporters, is like a technical *cult*, since it displays the following cult-like characteristics: a *sense of higher purpose*, *loaded language*, *an exclusive community*, *persuasive leadership*, *revisionism*, and an *aura of sacred science*.[39]

With regard to *loaded language*, Johnson argues that perhaps XP's most egregious effect on the broader software engineering community has been to "infect communication with cutesy slogans and acronyms" like "extreme" and "refactoring". As in any cult, Johnson argues, the purpose of jargon like this is to "elevate the mundane to the significant through relabeling, and to misdirect attention away from failings and inconsistencies in the cult's value system" (p. 60). This approach of the XP leaders to create a custom vocabulary for their followers resembles the way in which Kuhn used Wittgenstein's concept of *language games* to describe how the practitioners within different paradigms no longer speak the same language, in the sense that the rules of the words' usage have changed within the new

[39]The critique in this section is not intended to suggest that the XP methodology, because it displays cult-like characteristics, cannot be used successfully as a software methodology. In fact, its popularity to date may arguably be because of these cult-like characteristics, rather than in spite of them. One could argue that "great" software teams are recognised by the dedication to their mission—perhaps even a world-changing one—by their close working relationships, and by their special identity. Disney and Apple in the early years, the Skunkworks project at Lockheed, and even the Manhattan Project are examples of projects with "great" software teams, but which also perhaps displayed cult-like characteristics.

paradigm. Indeed, it was shown in section 3.1 that Wittgenstein's language games have encouraged some Agile methodologists—most notably one of the founders of Agile software development, Cockburn—to view the process of software development as the unfolding of a language game, in which new words are added to the language over time. The "rules of the game", which Kuhn uses to refer to the rules of the new paradigm can be compared to the rules of XP, which manifest themselves in the values, principles and practices of the methodology. Furthermore, the "language communities", which Kuhn uses to represent the strikingly different terminology used by the proponents of the new paradigm, can be compared to the new words that XP invented, or to the terminology that XP redefined. This use of custom vocabulary leads to a form of relativism that insulates the new XP language community from criticism by members of a competing paradigm, since the paradigms are no longer *commensurable*.

With regard to *revisionism*, Johnson points out that the most notable example of revisionism in XP can be found in the C3 project[40]. He argues that:

> Proponents of XP repeatedly use this project as their poster child, the tacit claim being that its success is evidence of the validity of XP. However, the reality is that the C3 project was a failure... XP advocates have chosen to cast this failure as a success, by carefully defining the criteria for success..." (p. 60)

There are clear parallels between the revisionism encouraged by the XP leaders, in this regard, and the historical revisionism, which Kuhn argues occurs when science textbooks are re-written by members of the triumphant scientific paradigm. Furthermore, the tendency of the XP community to interpret real world events in a way that confirms their existing biases, and to deny evidence to the contrary, is very similar to the way in which scientists, during normal science, refrain from attempting to refute hypotheses, theories or the paradigm. It is also known that the leaders of the XP movement have generated a large amount of literature which has, no doubt, led to widespread revisionism and also to the indoctrination of software engineers into the XP "paradigm".

Johnson argues that the *aura of sacred science* implies that the laws and tenets of the discipline are beyond question. Regarding XP, he points out that the twelve core XP practices are highly interdependent, which means that removing any one of them will

[40]See CCC (2008), and section 2.3 where the C3 project was discussed.

most likely result in the collapse of the whole or at least a shift from the 'purity' of the methodology. Perhaps this is why Beck & Andres (2000) (the *first* edition of *Extreme Programming Explained*), suggest that the XP practices be adopted *holistically*. More specifically, they write "The full value of XP will not come until all the practices are in place". Later in the book, however, it is stated that: "This poses a dilemma. Is XP all or nothing? Do you have to follow these practices to the letter or risk not seeing any improvements? Not at all. You can get significant gains from parts of XP. It's just that I believe there is much more to be gained when you put all pieces in place". Then, in the second edition, Beck admits:

> Critics of the first edition have complained that it tried to force them to program in a certain way. Aside from the absurdity of me being able to control anyone else's behaviour, I'm embarrassed to say that was my intention. Relinquishing the illusion of control of other people's behaviour and acknowledging each individual's responsibility for his or her own choices, in this edition I have tried to rephrase my message in a positive, inclusive way. I present proven practices you can add to your bag of tricks. (p. xxii)

Beck's concession in the second edition shows a clear theoretical shift away from an authoritarian approach towards a more liberal approach, regarding the adoption of the XP methodology. In part II of this dissertation, it was shown that the approach advocated in the first edition towards the adoption of XP, could be described in terms of Kuhn's notions of "paradigm shift" and "scientific revolutions", whereas the approach advocated in the second edition, could be described in terms of Popper's *evolutionary epistemology*. The impact on the software team of the incremental adoption of XP, and on the user community of the *incremental deployment* of software as a result, was also shown in the previous section of this chapter to resemble the impact on society of *piecemeal social engineering*.

Johnson argues that the all-or-nothing thinking, which is evident in the first edition of *Extreme Programming Explained*, is typical of cults and that it requires the members to display total dedication to the cause and to its objectives, or risk being labeled impure or being expelled from the community. As a result, members are discouraged from questioning the fundamental beliefs of the discipline, leading inevitably to a closed society. This all-or-nothing thinking, and Beck's original requirement of total dedication to XP agrees with

Kuhn's argument that a discipline becomes scientific only once all the practitioners adopt a single paradigm. The unquestioning nature that this approach encourages can be paralleled with the unquestioning attitude of Kuhnian scientists practising *normal science*, during which they resist counter-instances or anomalies until such time as the scientific community is plunged into a state of crisis. Furthermore, Johnson argues that often practitioners are forced to perform partial implementations of XP due to their organisational circumstances, for instance, and, if these applications are unsuccessful, the "failure is attributed to the impurity of their implementation rather than any failing or infeasibility of XP itself" (p. 61). In Kuhnian terms, the anomaly or counter-instance would be the failed project, which should discredit the paradigm—that is the XP methodology—but which, instead, is explained away as a misapplication of the methodology itself.

With regard to *persuasive leadership*, Johnson argues that the three leaders of the C3 project—Beck, Jeffries and Cunningham—have become "the XP community's ultimate arbiters of policy and direction". The XP community directs "questions about their own interpretations of the XP doctrine to these central figures, seeking their approval and advice". Therefore, "the knowledge of what is and isn't OK is seen to be held by a central authority". The authority that these XP leaders enjoy over the XP community can be compared to the epistemological authority senior scientists enjoy over new scientists in the paradigm. New XP team members learn from seasoned project team members through practices like pair programming, in a similar way that new scientists learn from the experienced members through tacit knowledge transfer and through hands-on experimentation, a process which Johnson regards as the "authoritarian distribution of sacred knowledge" from the experts to the novices. The danger to XP of this authoritarian approach is that it encourages the creation of blinkered and uncritical software engineers who are indoctrinated by the senior team members, much like the new scientists are controlled by the senior scientists in the new paradigm[41]. This approach also results in software engineers unaccountable to the public due to their assumed authority. Furthermore, the fact that the knowledge and decision-making are "not in the hands of the practitioners themselves" illustrates the undemocratic nature of this approach and the tendency towards a closed community. According to Fuller, contemporary science is not practised democratically, which

[41] Some of the techniques that senior scientists use are textbook revisionism and the double truth doctrine, i.e. propagating the myth that the new paradigm is the inevitable consequence of all preceding research in the search of truth whereas, in fact, the senior scientists know this to be false.

is reflected in Kuhn's paradigms:

> In contrast, as science has acquired more secular power, it has tended toward the self-perpetuation of existing regimes, as dominant research programmes are pursued by default... [Kuhn and Merton] saw science as mainly about those few who rise above the rest and constitute themselves as a self-perpetuating community... The only sense in which Kuhnian scientists dictate the terms of their own inquiry is that they all agree to abide by the decisions taken by their élite peers. This, in turn, provides a united front of legitimacy to the larger society. It should come as no surprise that Kuhn's only interest in the sociology of science lay in the acculturation of novices into a scientific paradigm, since thereafter the novice's mind is set to plough the deep but narrow furrow laid down by her senior colleagues as normal science. (pp. 51-52)

So far, this section has shown that certain leaders of the Agile and XP methodologies behave in ways that Kuhn associates with the behaviour of scientific communities. To this extent, Kuhn seems to be an accurate describer of the way science-like activity occurs. What is important to note, however, is that there is a definite tension between the way in which the XP leaders promote their methodology, and the avowedly democratic values, principles and practices underlying the methodology.[42] Kuhn's theory helps highlight these authoritarian tendencies.

Johnson also points out that "the argument from authority is everywhere in the Agile and XP communities..." and "not only are such appeals frequent, they are at the very heart of the rhetoric". He cites a particular blog entry entitled *AppealToAuthority* (Fowler, 2008) made by an influential Agile advocate, Martin Fowler, in which he concedes that he occasionally receives comments like "When a guru like you says something, lots of people will blindly do exactly what you say". Fowler replies that he does not believe software engineers would unthinkingly adhere to any advice he gives. Even if this may be so, the uncritical attitude that some Agile supporters seem to adopt, can be compared with the uncritical attitude of Kuhnian scientists practising normal science. Moreover, the adherence by the XP supporters to direct and indirect appeals to authority made by gurus in the XP field, can be contrasted with Popper's advocacy of an open society. Such appeals to authority

[42] For instance, the XP practice of *on-site customer* involves the layman in important decision-making, whereas the Kuhnian scientist is never accountable to the public and never consults with laymen.

by the XP gurus, according to Johnson, include their use of statements prefixed with "In my experience" to assert that their opinion is grounded in and supported by extensive experience; their use of sweeping statements and broad generalisations, which suggests that they have some kind of absolute factual knowledge; their quotation of new terminology and slogans, which give their statements an assumed, although often mistaken, authority; and their claims like "The XP community believes X", which suggests that their claims represent those of the group when, in fact, the group has often not been consulted, nor does the group necessarily agree with such claims.

With regard to the *creation of an exclusive community*, Johnson points out that the "fact that there is a community of enthusiastic proponents behind XP serves only to enhance its credibility via the principle of *social proof*". It certainly does not prove that XP is the panacea to the many problems of software engineering. Despite this fact, as Johnson notes, many of the XP advocates hold an arrogant and somewhat élitist attitude by thinking "that they have *got it*, and that anyone else not similarly enthused is a laggard". The same air of exclusivity and élitism is characteristic of the Kuhnian scientific community: it would be seen as absurd for any member outside the scientific community to question the findings of the scientists within the Kuhnian scientific community. Johnson's criticisms of the supporters of Agile and XP as exclusive communities—which Popper would describe as closed societies—can also be applied to the Agile leaders themselves, most of whom are, or have been, affiliated with the exclusive Cutter Consortium.

Johnson also contends that XP zealots have a *sense of higher purpose*, claiming to represent the most modern developments in software engineering. He detects in their claims an arrogance and élitist attitude, which, he believes, is not justifiable. While Kuhn makes no explicit reference to a *sense of higher purpose* in his *Structure*, certain adherents of XP, use Kuhn's ideas in such a way as to suggest a sense of higher purpose—for example, Yourdon's reference to "a band of renegade scientists, engineers, or priests looking for a new paradigm". Elsewhere Yourdon uses the word "soothsayers", which even more closely relates to a sense of destiny, since soothsayers are believed to have privileged access to the workings of destiny in the form of visions and prophecies. Perhaps XP is seen by many software engineers as the new religion of the software engineering community in which the XP leaders, especially Kent Beck, are seen as the élite, authorities holding the mysteries to the XP methodology. This conjecture finds corroboration in Fuller's contention at the start

of this chapter that "It turns out that religious conversion is Kuhn's model for the paradigm switch that occurs during a scientific revolution". By analogy, the shift to Agile and XP methodologies can be understood as a religious conversion or "leap of faith". All of these examples are typical of a closed and authoritarian society, and are opposed to the critical, scientific spirit of Popper's open society. Popper may have criticised this sense of higher purpose as an example of historicism, just as Popperians have criticised Kuhn's concept of science as historicist, since the new paradigm's success is seen as somehow inevitable and its authority unquestionable until it, in turn, is displaced. This is similar to ethical positivism, the belief that the laws of the day are right simply because they are the dominant laws: it collapses the distinction between 'is' and 'ought', just as Kuhn does, and subscribes to a form of 'might is right'.[43]

In another chapter of his book called "XP and ESP: The Truth is Out There!", Johnson argues that there are two sorts of bias, *positive outcome bias* and *confirmation bias*, which together make it possible for XP advocates to sustain the false belief that XP has been "proved" in the field. Regarding *positive outcome bias*, Johnson claims that "If there is sufficient desire to find 'evidence' favorable to XP, the XP practitioners will find it. Furthermore, if there is sufficient reward for publication of XP success stories, they will be published". The latter criticism is corroborated by Carroll (2003), who argues that there is a tendency of researchers and journal editors to publish research with positive outcomes much more frequently than research with negative outcomes. This is perhaps not surprising since success stories would tend to sell better, and publishers, driven by capitalism, have first and foremost the pursuit of their own profit in mind. Publishers would then be guilty of *positive outcome bias* and of "re-writing the textbooks", just as Kuhnian scientists re-write the textbooks when a new paradigm overthrows an old one. A consequence of the tendency to publish research with positive outcomes may be that most of the negative outcomes of XP are never published. Instead, 'XP success story' literature continues to be published predominantly for the profit it brings. Therefore, it can be conjectured that the influence of XP on the software engineering community may, at least partly, be explained by the prolific literature concerning the successes of XP rather than on case studies of projects, which have tried and tested the XP methodology.

[43]The sense of higher purpose that Johnson perceives in Agile may be related to the ideology of 'Manifest Destiny' that pervades American views on their supposed historical purpose in the world.

In order to propagate evidence of the successes of XP, advocates will often choose to ignore conflicting evidence and to censor cases of failure. These tendencies have direct parallels with Kuhnian *normal science*, during which scientists resist counter-instances and explain away failures of the paradigm, willfully ignoring any contrary evidence. Kuhnian scientists are so committed to the paradigm, to the entire constellation of beliefs, values and techniques shared by the members of the scientific community, that they have a tendency towards noticing and indeed searching for confirming instances of their beliefs, and self-censoring any negative evidence which may contradict these beliefs. *Confirmation bias* is also evident in XP, according to Johnson, if one considers how the XP community handles case studies from the field: "When presented with a claim of success using XP, the community accepts it without challenge, for it is a welcome confirmation of pre-existing beliefs. However, a claim that XP has failed is an unwelcome affront to their personal convictions. So these claims are scrutinized until an 'out' is found..." (p. 73). This confirmationist approach is typical of that adopted by Kuhnian scientists since they share much in common with logical positivism despite Kuhn's critique of that philosophy. Finally, Johnson concludes "The XPers have all their bases covered. No matter what the experience report, there is no need to ever cast doubt upon XP itself—there are always rival causes to be blamed". To use Popperian terminology, this belief ultimately means that the XP methodology becomes *unfalsifiable*.

By showing in this section that several authoritarian aspects of XP, which Johnson criticises, have parallels in Popper and Fuller's criticisms of Kuhn's philosophy, an awareness has been created and a means can be devised to counter these authoritarian tendencies. To conclude this section and to reiterate the dangers of a closed society, it is interesting to note that the *Hacknot* website, along with the online book and resources, has recently become unavailable on the Internet. The reason for this is not known but one could speculate that the website has been censored due to its critical nature. Perhaps Johnson's own words at the start of his book are somewhat prophetic:

> Many of the essays on Hacknot take a stab at some sacred cow of the software development field, such as Extreme Programming, Open Source and Function Point Analysis. These subjects tend to attract fanatical adherents who don't take kindly to someone criticizing what for them has become an object of veneration...

Chapter 7

Other Influences on the Agile Ethos

The aim of this chapter is to investigate the influence of the architect *Christopher Alexander*, and the authors of *Peopleware*, *DeMarco and Lister*, on the ethos of the Agile and XP software methodologies, and to investigate whether aspects of their approaches resemble aspects of Popper's *piecemeal social engineering* and *open society*.

On the one hand, Alexander, whose architectural "pattern discipline" was adopted by members of the software community in the 1980s, inspired the *design patterns movement* in software engineering, and later influenced the XP movement (CAL, 2008). On the other hand, *DeMarco and Lister's* influential 1987 book called *Peopleware*, was one of the first in the field to raise an awareness that sociological aspects are just as important—and sometimes even more important—than technological aspects in software development. Alexander and the Peopleware authors have not been chosen for inclusion in this chapter merely because of their link to the Agile and XP methodologies, but also because it is conjectured that there may be a strong affinity between Alexander's ideas on *social engineering architecture* and Popper's notion of *piecemeal social engineering*, as well as many similarities between the ideas advocated by the *Peopleware* authors concerning *management style* in software engineering, and those underlying Popper's notion of an *open society*. Furthermore, DeMarco and Lister explicitly use the ideas of Alexander in their work: Alexander's *concept of organic order* appears in a chapter of the second edition of *Peopleware* in 1999 relating

to "the office environment"[1], and, more recently, Alexander's ideas concerning patterns in architecture inspired DeMarco and Lister's 2008 book called *Adrenaline Junkies and Template Zombies: Understanding Patterns of Project Behaviour* (DeMarco *et al.*, 2008). In that book, DeMarco, Lister, and their coauthors, extend the *design patterns movement* in software engineering, which has until now concentrated entirely on patterns of the software itself, towards patterns of projects and teams. In other words, their book focuses on the *sociological* aspects of patterns rather than on the technological aspects.

The first section of this chapter will discuss Alexander's ideas and will highlight their influence on the software engineering discipline. Then it will investigate the extent to which Alexander's *social engineering architecture* resembles Popper's *piecemeal social engineering*, and whether Alexander's concept of a *meta-plan* can be paralleled with XP's approach of *incremental design*.

The second section of this chapter will investigate whether the ideas of DeMarco and Lister are aligned with Popper's notion of an *open society*, by analysing briefly each of the chapters in *Peopleware*.

7.1 Christopher Alexander's *Social Engineering Architecture*

Christopher Alexander is a highly respected and influential *architect* "noted for his theories about design, and for more than 200 building projects in California, Japan, Mexico and around the world" (CAL, 2008). Alexander believes that the architect should be held accountable to the client (similarly to the way the state should be held accountable to its citizens in an open society), and that the client should be able to criticise the architect's design before it is implemented, because the client usually knows more about what he/she requires in a building than the architect does. Alexander argues that there are several competing forces—including cost, speed, quality, aesthetics, function, utility, and so on—represented by different stakeholders on a building project. To counter these competing forces, and balance the power[2] between different stakeholders, Alexander created his *pattern*

[1] In fact, it was this part of *Peopleware*, where Alexander is mentioned, that inspired the topic of this dissertation.

[2] Indeed, Popper would also insist on a balancing of power rather than merely shifting authority from one stakeholder to another.

language—a practical architectural system—to empower any person to design and build spaces for themselves at any scale.

Alexander's book called *A Pattern Language: Towns, Buildings, Construction* (Alexander *et al.*, 1977) impacted a generation of software engineers in the 1980s, and is now widely recognized to have inspired the *design patterns movement* in software engineering[3]. The most important concept in that book, according to Alexander, is his *idea of diagrams* (or what he later termed *patterns*), and which he defines as:

> an abstract pattern of physical relationships which resolves a small system of interacting and conflicting forces, and is independent of all other forces, and of all other possible diagrams.

Besides the influence of *A Pattern Language*, Alexander's *Notes on the Synthesis of Form*, which was prescribed reading for researchers in computer science throughout the 1960s, is said to have influenced many aspects of the software engineering discipline, including programming language design, modular and object-oriented programming, and other design methodologies. With regard to XP specifically, CAL (2008) claims that Alexander's advocacy "of incremental, organic and coherent architectural design also influenced the XP movement". Indeed, Beck & Andres (2005) acknowledge Alexander and entitle one of the chapters of their book "The Timeless Way of Programming"—a tribute to Alexander's 1979 book, *The Timeless Way of Building* (Alexander, 1979). In that chapter, Beck argues that the same imbalance of power, which Alexander identifies in his discipline between the architect and the client of a building, often exists in software development too, between the programmer and the customer. This explicit influence of Alexander's democratic ideal on XP, may account, at least partly, for the seemingly democratic values, principles and practices of the XP methodology.

The aim of this section is to investigate whether certain of the values underlying Alexander's *social engineering architecture* are fundamentally aligned with certain of the values underlying Popper's *piecemeal social engineering* in an *open society*. If such fundamental

[3]It is interesting to note, however, that Alexander (1964) explicitly and publicly states in the preface to the reprinted version of his original book that "I reject the whole idea of design methods as a subject of study, since I think it is absurd to separate the study of designing from the practice of design".

similarities can be found, it can be conjectured that these similarities paired with Alexander's influence on the XP leadership, may account, at least in part, for the apparent Popperian spirit of many of the XP values, principles and practices (as detailed in chapter 5). In making this conjecture, it is important to note that Alexander is aware of Popper's philosophy. In his *Notes on the Synthesis of Form* (Alexander, 1964), Alexander makes three explicit references to the works of Popper. Firstly, in the introductory chapter "The Need for Rationality", he criticises architects who cling to intuition in the face of logic and reasoning in order to shelter themselves from the loss of innocence. He parallels this with the closed society that Popper criticises in his *Open Society and Its Enemies*, in which citizens cling to intuition in order to resist open criticism. Alexander believes that logic, instead of merely intuition, is essential in architectural design due to the growing complexity of architectural structures. Secondly, in a chapter entitled "Definitions", Alexander argues that poorly designed artefacts lead to stress, which often result in *misfit*. While describing the approach of eliminating such misfits[4], Alexander cites the similar negative approach to ethics taken by the Popperian piecemeal social engineer, namely, to "Minimise suffering" by searching for and fighting against the most urgent tensions in society, rather than aiming to achieve some vague and undefinable notion of an "ultimate good". The third reference Alexander makes is to Popper's contribution to physics (rather than his philosophy of ethics). He cites Popper's *The Propensity Interpretation of the Calculus of Probability, and the Quantum Theory* when discussing causality in architectural structures.

Alexander's *Social Engineering Architecture* and Popper's *Piecemeal Social Engineering*

An important step towards resolving the competing forces between different stakeholders of a project is, according to Alexander, the *rejection of a master plan* for architecture. This would mean that the architect can no longer impose a design on his client. Alexander's rejection of a master plan for architecture resembles Popper's rejection of a master plan for social engineering. Instead of a master plan, Alexander proposes a *meta-plan*—"a philosophy by which a facility can grow in an evolutionary fashion to achieve the needs of its

[4]The example Alexander uses to describe his theory of "misfit variables" is how to ensure a surface is flat: ink a known flat surface and rub it on the target surface; those places with ink left on them are the "high" areas, which need to be planed down.

occupants" [5]—which constitutes three parts:

1. a philosophy of piecemeal growth;
2. a set of patterns or shared design principles governing growth; and
3. local control of design by those who will occupy the space.

Under Alexander's meta-plan, communities grow in an evolutionary fashion through a series of small steps, in an ongoing and iterative process, which results in what Alexander calls *organic order*[6]. The three parts of Alexander's meta-plan can be compared to the approach of XP software development. The meta-plan's "philosophy of piecemeal growth" resembles XP's "philosophy of incremental design", its "set of patterns or shared design principles governing growth" resemble the software design principles in the *The Agile Manifesto*, and the "local control of design by those who will occupy the space" has a parallel in the XP practice of *informative workspace*.

In his three-volume work, *The Timeless Way of Building*[7], Alexander (1979) promotes a philosophy of incremental, organic and coherent architectural design by which "facilities evolve through a series of small steps into campuses and communities of related buildings"[8]. Consequently, Alexander would agree with the piecemeal social engineer who, according to Popper, recognises that:

> only a minority of social institutions are consciously designed while the vast majority have just 'grown', as the undesigned results of human actions. (Miller, 1983)

Alexander's approach to *social engineering architecture* and Popper's approach to *piecemeal social engineering* are both evolutionary and democratic, rather than utopian and authoritarian. Neither Alexander nor Popper attempt to fulfill a blueprint at all costs, but instead they remain flexible and accountable to the needs of individuals. Popper's principles of social openness, democracy, criticism and rationalism[9] seem also to be inherent in Alexander's

[5] DeMarco & Lister (1999) (p. 83)

[6] The notion of organic order should not be confused with organic theories of the state or with the organic order of closed societies, as pointed out by Popper (1966) (pp. 173-174). In an organic state, the individual is completely subordinate to the whole.

[7] In the *Timeless Way of Building*, Alexander ties life and architecture together. The book is considered by many to be one of the seminal texts of the 20^{th} century (TWB, 2008). It can be said that Alexander's central aim in the *Timeless Way of Building* is to bring a more humane and democratic character to the modern cityscape.

[8] DeMarco & Lister (1999) (p. 83)

[9] Alexander's rationalism is evident especially in his replacement of intuition with logic and reason.

approach to architectural design.

In addition to the strong affinity between Popper's *piecemeal social engineering* and Alexander's *social engineering architecture*, one finds in Alexander's criticism of full-scale architectural engineering, many similar points that Popper raises in his criticism of utopian social engineering.[10] According to Alexander, the development of a master plan for architectural design is the first and fatal deviation from the *Timeless Way of Building*. Alexander argues that a master plan envisions hugeness and grandeur, and encourages replication to achieve an enormous whole of identical components, which results in sterile uniformity. Alexander observes: "master plans are incomplete for the first 25 years and obsolete thereafter" (MPL, 2008). Popper would no doubt have agreed with Alexander's statement cited by DeMarco & Lister (1999) (p. 83):

> The master plan is an attempt to impose totalitarian order. A single and therefore uniform vision governs the whole. In no two places is the same function achieved differently. A side effect of the totalitarian view is that the conceptualization of the facility is frozen in time.

Furthermore, in a similar way that Popper reminds scientists of their accountability to society, Alexander, in a landmark keynote speech at OOPSLA in 1996, challenged software engineers to be mindful of their ethical responsibilities to the community and to act on these responsibilities. In particular, he reminded the conference attendees to be aware of the ends to which their software will be used, encouraging them to see themselves as moral agents and not merely as technicians or instruments providing technical expertise for any ends whatsoever. Alexander's moral challenge to the software engineers during this keynote speech, can be summarised as follows: *As the creators of the emerging virtual world, software engineers should endeavour to ensure that the software they produce will be used to make the world a more humane place.* Alexander challenged the software engineers to build software systems that contribute powerfully to the quality of the lives of citizens, and to recognize and rise to the responsibility that accompanies their current position of influence in the world, as creators of the emerging virtual world. Alexander (1999), the published version of Alexander's keynote speech, is introduced by Coplien:

> Once in a great while, a great idea makes it across the boundary of one discipline

[10] Note that the metaphor "software architecture", which is widely used in the jargon of the software community, is actually derived from the traditional building-crafts.

> to take root in another. The adoption of Christopher Alexander's patterns by the software community is one such event... It is odd that his ideas should have found a home in software, a discipline that deals not with timbers and tiles but with pure thought stuff, and with ephemeral and weightless products called programs. The software community embraced the pattern vision for its relevance to problems that had long plagued software design in general and object-oriented design in particular. Focusing on objects had caused us to lose the system perspective. Preoccupation with design method had caused us to lose the human perspective. The curious parallels between Alexander's world of buildings and our world of software construction helped the ideas to take root and thrive in grassroots programming communities worldwide. The pattern discipline has become one of the most widely applied and important ideas of the past decade in software architecture and design... The timing and audience of the venue afforded Alexander the chance to reflect on his own work and on how the object-oriented programming community had both hit and missed the mark[11] in adopting and adapting his ideas to software. As such, the speech was a landmark event that raised the bar for patterns advocates, for object-oriented programmers, and for software practitioners everywhere. Beyond that, this speech has timeless relevance to any engineering, scientific, or professional endeavor.

In summary, this section has shown that the approach of Alexander's *social engineering architecture* is fundamentally aligned with the approach of Popper's *piecemeal social engineering*, and that Alexander's critique of *full-scale architectural engineering* resembles Popper's critique of *utopian social engineering*. It also pointed out that Alexander makes explicit references to several of Popper's works. Furthermore, it was shown that Alexander's ideas influenced the leaders of the XP movement, most notably Kent Beck. These links are relevant to the argument in this dissertation, since they suggest that Alexander's work provides a causal link between Popper's ideas and those of XP. *This may at least partly explain the apparently Popperian spirit of many of the XP values, principles and practices.*

[11] Note here the parallels with a fallibilist approach, as Popper advocates.

7.2 DeMarco and Lister's *Peopleware*

DeMarco and Lister published the first edition of their popular software management book—*Peopleware: Productive Projects and Teams*—in 1987, and the second edition in 1999[12]. The discussion in this section relates specifically to the second edition of *Peopleware*. Many consider *Peopleware* to have "started a revolution" in software engineering mainly because it was one of the first books of its kind to demonstrate that the major issues of software development are *human*, not technical, emphasising that "Sociology matters more than technology or even money" (p. 174). This is the first reason *Peopleware* has been chosen for inclusion in this section—to highlight the social concerns (from which ethical implications often follow) that *Peopleware* raised in the previous decade, concerns which are arguably just as relevant to the software community today[13]. Another reason for including *Peopleware* in this chapter, is to investigate the similarity between DeMarco and Lister's ideas regarding *management style* in software development communities, and the ideas underlying Popper's notion of an *open society*. A final reason for including *Peopleware* here, is its links to Christopher Alexander's work, to Agile methodologies, and to XP in particular. DeMarco and Lister are also both presently consultants for *Agile Software Development and Project Management* at the Cutter Consortium, which means their ideas continue to have an influence on the various Agile methodologies, as well as on the Agile community more broadly. Their influence on XP, in particular, is also documented, most notably by Beck & Andres (2005) in the annotated bibliography of their book where they state, for example, that *Peopleware* was their source for the principle of 'accepted responsibility'.

Before investigating the six parts of *Peopleware*, the following points are important to note. The title of DeMarco and Lister's book—*Peopleware*—seems to contain a tension

[12]Other such books, which were published prior to *Peopleware*, include Gerald Weinberg's *The Psychology of Computer Programming* (Weinberg, 1971), and Fred Brooks' *Mythical Man Month* (Brooks, 1975). In the 25^{th} anniversary edition of Mythical Man Month, Brooks points to *Peopleware* as the influential IS book of the 1980s, and says that it was influential for the same reason Mythical Man Month was: The primary challenges of software development are social, not technical. In a panel session celebrating the 20^{th} anniversary of the publication of *Peopleware*, Brooks called DeMarco and Lister's book: "the best book on software management to appear in the last twenty years" (Fraser *et al.*, 2007). McConnell hailed the original publication of *Peopleware*, a watershed event, and pointed out that while the term "peopleware" had already been coined in 1976 by P.G. Neumann, it was not until *Peopleware* was first published in 1987, that a compelling case for elevating people-centered considerations to center stage, was presented.

[13]See Yourdon (2007) in which Boehm predicts that *Peopleware* will survive a long time since the stories are about people, and they are just as true today as they were when the book was first published.

between a Kantian ethic of respect for persons—the "*People*" of the title—and a utilitarian instrumentalist ethic—the "*ware*" of the title, which, it will be argued, is not entirely resolved in the book. On the one hand, the authors are concerned that software developers should not be treated merely as means to the ends of the companies they work for. On the other hand, the book is concerned with increasing productivity as an non-negotiable goal of software development (no doubt with competing companies in mind). The underlying argument seems to be that software developers will be more effective (hence more efficient means to the ends of the software company) if they are treated more humanely, namely, respected as persons. The word "ware", meaning "articles of the same type" (and merchandise), sits uncomfortably with the personhood implied by the word "people".[14] Another strand of the book, which coincides closely with Popper's open society, is its clear stand against authoritarian *managerial styles* and its plea for openness, flexibility and a willingness to experiment, and learn from errors. By making these points—as well as the capitalist framework—explicit from the start, it is hoped that the ethical and ideological "deep structure" of *Peopleware* will be clearer in the rest of the discussion.

In the first part of *Peopleware*, which deals with "Managing the human resource", the authors question the *management style* of those managers who treat programmers as resources, as though they are modular components—merely means to the companies ends. Similarly, Beck criticises the tendency of large organisations to abstract people to things, plug-compatible programming units. He argues in *Extreme Programming Explained* that large organisations "often ignore the value in teams, adopting instead a molecular/fluid metaphor of 'programming resources'. Once a project is done they go back 'into the pool' ". The *Peopleware* authors argue that a very different way of thinking about and managing people needs to be considered, which will accommodate the very non-modular character of human beings. By conducting surveys each year of failed projects, the authors discovered that for the majority of the failed projects "there was not a single technological issue to explain the failure" (p. 4). Instead, the results suggested that the nature of the problem was the project's *sociology*, which led the authors to conclude that "The major problems of our work are not so much technological as sociological in nature" (p. 4).

[14]Of course, the *Peopleware* authors are merely trying to make the most of the "given" of an increasingly competitive, Tayloristic and capitalistic economic framework, which places pressure on the freedom and dignity of the individual. While this is my own interpretation, others may argue that the title *Peopleware* is tongue-in-cheek, showing that people should not be treated the same way as the different 'wares' in software, hardware, middleware, firmware, and so on.

In relation to this observation, the *Peopleware* authors criticise several aspects of software development and management, many of which are arguably still valid in software development today, including:

- The tendency of managers to view the software development process as a production environment. They argue that this leads to treating workers as interchangeable pieces, to ignoring the uniqueness of each individual, and to the discouragement of experimentation. To counter the latter tendency, the authors encourage software engineers to experiment, and not to be afraid of making mistakes, an approach which resembles that of the piecemeal social engineer who makes small, experimental changes in society in the knowledge that these changes can be reversed if found to be detrimental. Beck & Andres (2005) would also agree with this view since they argue that value in software is created not just by the 'resources' that people have—what they know and do—but also by their relationships, and what they accomplish together as teams. Furthermore, Beck states that "Software development is a human process not a factory".

- The tendency of managers to adopt a style of management defined by: 'Management is kicking ass'. The *Peopleware* authors argue that it is seldom necessary to take Draconian measures to keep people working. They believe that an authoritarian approach discourages innovation and creativity, and the free flow of information. This critique of an authoritarian management style resembles Popper's critique of authoritarian governments and regimes.

- The tendency to emphasise *productivity* at all costs. The authors argue that this tendency leads to pressurising people to work excessive amounts of overtime due to hopelessly tight schedules, and to a compromise in quality (which, in turn, lowers the self-esteem and job satisfaction of the employee). These consequences also often increase staff turnover. While the *Peopleware* authors do not question increasing productivity as the overriding imperative, they argue that programmers will be more productive if their individuality is acknowledged and respected.

The *Peopleware* authors conclude the first part of their book by stating "The manager's function is not to make people work, but to make it possible for people to work" (p. 34).

This statement bears further testimony to the anti-authoritarian approach, which these authors are committed to.

The second part of *Peopleware* deals with "The Office Environment". The authors argue that working conditions have degraded significantly over the past decade so much so that the modern day office environment is plagued by several types of interruptions. These interruptions, they argue, fill the employees' days with frustration and significantly reduce their efficiency. They note that "the environmental trend is toward less privacy, less dedicated space, and more noise" (p. 51), all in the name of reducing cost[15].

The *Peopleware* authors' concerns for how the office environment impacts the people who work in it, suggests their adherence to a Kantian respect for persons, not simply a respect for the individual's unique needs, but, more importantly, for the individual's rational autonomy. If software engineers are regarded as unique individuals, they will respond more responsibly (and no doubt be more productive too;). These concerns of the Peopleware authors were influenced by Christopher Alexander, the architect whose ideas were discussed in the previous section of this chapter. Ultimately, the authors argue that we should aim towards a workspace that has certain time-proven characteristics, as Alexander advocates in his book, *The Timeless Way of Building*. Through his influence on DeMarco and Lister, Alexander has influenced the Agile community, fostering an awareness amongst them of humane working environments. As pointed out in the previous section too, XP was also influenced by Alexander in this regard, and the XP practice of *Informative Workspace* may well have been inspired by Alexander's ideas.

In the third part of *Peopleware*, which is entitled "The Right People", the authors' main argument is that the "final outcome of any effort is more a function of *who* does the work than of *how* the work is done" (p. 93). This statement, at first sight, seems to be élitist. Indeed, the authors are aware of the tension this belief creates when they write "In our egalitarian times, it's almost unthinkable to write someone off as intrinsically incompetent.

[15]While this may seem anachronistic, and we should remember that "the past decade" refers to the 1990s (since the second edition of *Peopleware* was published in 1999), I can personally vouch for the contemporary reality of a noisy and disruptive office environment. At the large multi-national corporation where I worked for three years, cubicles were separated merely by grey barriers, which provided little privacy (since the partitions did not reach the ceiling), and virtually no protection from noise. My day was continuously interrupted by phones ringing off the hook and the conversations of people (including those of passers-by, those on phones, and those of fellow colleagues). On most days, I used headphones to drown out the noise in order to work effectively. Other interruptions included daily meetings and teleconferences (often several such meetings per day) and the receipt of up to 100 emails per day. All these disturbances led to a highly disruptive working environment, one in which sustained concentration was virtually impossible.

There is supposed to be inherent worth in every human being" (p. 96). However, the authors argue that this view is not realistic when applied to software teams. To justify their position, they argue that it is all-important to get the right workers in the first place because managers are unlikely to change their workers in any fundamental way, especially since workers rarely stay at a company long enough. Therefore, they argue, if the workers are not right for the job from the start, they never will be. In this regard, the authors seem to have muddled "inherent worth" up with "inherent abilities" or with "innate capacities". As Kant argues, each person *does* have inherent value or worth, irrespective of the person's instrumental value. What this shows is that the Peopleware authors are still judging the "worth" of workers in terms of their value as means to the company's ends, despite their strong message in the first part of the book to the contrary. In addition, they extend their preoccupation with the "best workers" to a preoccupation with the "best organisations".

Notwithstanding the authors' questionable view regarding "The Right People", they criticise several aspects of authoritarian organisations towards the end of the third part of *Peopleware*. Firstly, they criticise companies who aim to ensure uniformity by, for instance, insisting on a dress code. According to the authors, making everything uniform, suits only the "owner" of the workspace who can demonstrate control over the people who inhabit the space. This emphasis on uniformity, lack of privacy, and authority are all symptomatic of the collectivistic and authoritarian ethic of a closed society. In my experience, some managers even go so far as to insist desks are kept tidy, especially when they expect a visit from senior management. Some companies also have an internal "computer security" department, which ensures policies are in place to prevent employees from installing software on their workstations not approved by the internal department. This ensures further that workstations are kept uniform. The authors also criticise organisations, which regard workers as *professionals* only on the condition that they look, act, and think like everyone else. Furthermore, the authors cite "the company move" as an insecure manager's naked exercise of power, which disregards most individuals in the workforce. They also make the distinction between Big M Methodology and small m methodology[16], where Big M Methodologies are imposed on workers by the Methodology builders and are characterised by standardisation, document uniformity and managerial control. This exercise of control

[16]The distinction between Big M Methodology and small m methodology can be paralleled with the distinction this dissertation has made between traditional and Agile software methodologies.

covertly suggests that managers believe their workers are incompetent and need stringent methods to control their work. All the above criticisms point to a common tendency of authoritarian organisations to undermine the individuality, uniqueness and dignity of their workers, which ultimately have ethical implications for all involved.

The fourth part of *Peopleware* discusses "Growing Productive Teams" and provides some advice for facilitating the formation of successfully bonded teams. Many of the suggestions in this part of the book can be considered 'remedies' for the shortcomings of authoritarian organisations, which were raised in the third part of the book. The authors argue that while it is extremely difficult to provide a list of sure-fire ways for guaranteeing the formation of "jelled" teams, it is simple enough to provide a list of ways to inhibit the formation of teams and disrupt project sociology. Such a list, which would result in what the authors call *teamicide*, includes: *defensive management*, the tendency of managers not to trust their workers and to guard against mistakes their workers may make by physically overseeing their work; *bureaucracy*, often manifested in mindless paper pushing; *physical separation*, preventing close interaction; *fragmentation of time*, resulting in a continuous shift of focus for workers involved in more than one project; *quality reduction of the product*, which undermines the workers' self-esteem and work enjoyment; *phony deadlines*, which are impossible to meet, implying that success is impossible to achieve; *clique control*, where managers take steps to break up closely knit teams, usually because they themselves are excluded from such teams and feel threatened by them; *overtime*, which often destroys team formation due to team members' differing abilities to work overtime; and *motivational accessories*, which send a simplistic message that virtues can be improved by motivational accessories alone. All these forms of *teamicide*, according to the authors, cause damage by demeaning either the workers themselves or the work they do. What is striking about this negative approach to achieving "jelled" teams is just how strongly it resembles Popper's negative approach of error elimination.

In order to counter *teamicide* and especially *defensive management*, the authors recommend an "open kimono" style of management[17], which bears strong resemblance to Popper's *open society*. Some principles of this approach include respecting the autonomy of the workers, giving the workers the responsibility to do the job they've been assigned to, and trusting

[17] An "Open Kimono attitude is the exact opposite of defensive management. You take no steps to defend yourself from the people you've put into positions of trust." (DeMarco & Lister, 1999) (p. 144)

them to get the job done. An "open kimono" style of management would provide satisfying closure, would allow and encourage heterogeneity (as in a pluralistic society), would foster uniqueness in workers, and would give workers a voice in team selection. In turn, these principles would lead to a *natural authority* working in all directions in the organisation: managers are natural authorities in areas such as negotiating and hiring, whereas workers are natural authorities in areas concerning technical decision-making and estimating the time to complete a job. This "open kimono" style of management would result in an organisational culture similar in many ways to the open society that Popper advocates.

In the fifth part of *Peopleware*, entitled "It's Supposed to be Fun to Work Here", the authors criticise managers who believe it is their job to prevent their workers from having fun on the job. In order to instill a sense that work should be fun, the authors encourage: *experimentation* by recommending that workers embark on pilot projects, *competition* by playing games, and *creativity* and *openness* through brainstorming sessions, all of which, have parallels in the *open society* Popper advocates.

In the final part of *Peopleware* entitled "Son of Peopleware", the authors broaden their focus to the social aspects of entire organisations (in contrast to the focus of the rest of the book, which is on the social aspects of the development work within these organisations). They argue that most people fiercely resist *change*, of whatever kind, and that the fundamental response to change is not logical but *emotional*[18]. This is an important consideration since, as software engineers, we build and deliver systems, which have an impact on the people who use them, often redefining the way people work entirely or causing the people to change their work habits[19]. Therefore, from an ethical point of view, it is important to consider how such changes should be instituted. If they are simply imposed by an authority on the affected people, without consultation, they may very well have detrimental

[18]This contention of the *Peopleware* authors corresponds, in part, with Fuller's observation (detailed in section 6.2) that certain irrational factors are involved in adopting a new scientific paradigm. The *Peopleware* authors propose that we should view change in the way an expert in family therapy, Virginia Satir, does. The "Satir Change Model" involves four stages, namely, *old status quo*, *chaos*, *practice and integration*, and *new status quo*. Part of the reason change is so emotional, according to Satir, can be explained by the stage of *chaos*, which resembles the stage of "crisis" in Kuhn's theory. Both stages cause considerable distress. Firstly, it is frustrating and embarrassing to abandon familiar practices only to become a novice again, and, secondly, the stage of chaos is often mistaken for the *new status quo*, which means people think they are worse off than before. However, Satir emphasises that "Chaos is absolutely necessary, and it can't be shortcut" (DeMarco & Lister, 1999) (p. 200).

[19]Beck raises the same point in arguing that the adoption of XP may impact the entire organisation because "XP can facilitate shifting the constraint out of software development completely, so XP will have ripple effects throughout an organization applying it" (Beck & Andres, 2005) (p. 89).

consequences. The *Peopleware* authors also point out that a safe environment is necessary for change to occur, one in which people "will not be demeaned or degraded for proposing a change, or trying to get through one" (p. 200). This, too, reminds one of Popper's point that an open society should positively nurture criticism and actively protect the freedom to criticise. Taking *Peopleware* a step further, a fallibilist attitude to error elimination and experimentation should also be nurtured, meaning that organisations should welcome criticism so that problems can be identified and solved early.

In the penultimate chapter of *Peopleware*, the authors criticise managers who waste people's time by arriving late for meetings, who call 'ceremonial' meetings, and so on, thereby demonstrating a lack of respect for their team members, and asserting their authority. In the final chapter of *Peopleware* entitled "The Making of Community", the authors digress briefly into the world of Aristotelian philosophy in an attempt to describe how the best managers facilitate the formation of 'community'. They argue that "Aristotelian Politics[20] is the key practice of good management" (p. 224). Their choice of philosopher is somewhat surprising, since the software community that they describe, has been shown to be more aligned with an open society, than with Aristotle's *eudamonian* virtue ethic, which presupposes a privileged, leisured class of gentlemen, which allows members to pursue virtue precisely because they do *not* have to work for a living, unlike most software engineers.[21]

[20] Aristotelian philosophy, according to the authors, comprises five interdependent sciences, namely, *metaphysics*, *logic*, *ethics*, *politics*, and *aesthetics*.

[21] A similar criticism, regarding the choice of Aristotle's philosophy, can be leveled at Himanen, the philosopher of the so-called "hacker ethic", whose ideas are discussed in Appendix F.

Reflection: Part III

In chapter 6, it was shown, on the one hand, that the *XP team, as envisioned by Beck, resembles an "open society"*, since many of the XP values advocated by Beck, such as openness, honesty, accountability, respect, courage, and individualism, are also essential to the open society Popper advocates. It was also shown that XP's approach of *incremental design*, with its rejection of an overall blueprint for design, resembles Popper's approach of *piecemeal social engineering*, and that both approaches share a common focus on the *present* rather than the future. Furthermore, the approach of traditional *Waterfall-like* methodologies was shown to resemble utopian (or holistic) social engineering. On the other hand, using Johnson's critique of the XP methodology as a basis, it was shown that there are several parallels between Kuhn's notion of "paradigm shift", and the holistic and irrationalist way in which the XP methodology is promoted and adopted. From these findings in chapter 6, it can be concluded that XP has elements of both open and closed societies. An important distinguishing factor, in this regard, was the finding that the open society elements in XP relate mainly to the *theoretical* values, principles and practices of the methodology, whereas the closed society elements relate mainly to the *adoption* and *propagation* of the XP methodology by several of its leaders.

In chapter 7, it was shown, firstly, that Alexander's approach of *social engineering architecture*, with its *rejection of a master plan* for architecture, closely coincides with Popper's approach of *piecemeal social engineering*. Both approaches were found to be evolutionary and democratic, rather than utopian and authoritarian, and both authors were found to criticise full-scale engineering in their respective disciplines. Through Alexander's influence on the software engineering discipline, the investigation identified an important causal link between Popper's ideas and those of XP. Secondly, the ideas of the *Peopleware* authors, Demarco and Lister, regarding an "open kimono" *management style* were shown

to correspond with many of the values underlying an *open society*, as advocated by Popper, and the authors' criticism of authoritarian managerial styles, was shown to coincide with Popper's criticisms of *utopian social engineering*.

Chapter 8

Conclusion

This dissertation has used the contrasting ideas of two leading philosophers of science of the 20^{th} century, *Karl Popper* and *Thomas Kuhn*, to approach a philosophical understanding of the so-called *Agile* software methodologies. In particular, this dissertation had two aims: to provide a philosophical understanding, firstly, of the *theoretical* values, principles and practices underlying Kent Beck's *Extreme Programming (XP)*—arguably the most prominent Agile methodology—and, secondly, of the *promotion* and *adoption* of the XP methodology as an alternative to the more *traditional* software methodologies.

Part I presented the context for this dissertation, by providing, on the one hand, an *historical* account of both scientific and software methodology, and, on the other hand, a description of the work related to the topic of this dissertation. It was shown that, while certain software engineers have used the ideas of various philosophers to illuminate their discipline, Kuhn's ideas have not been used critically, nor Popper's systematically, to provide a philosophical understanding of Agile software methodologies—hence the topic of this dissertation.

Part II of this dissertation dealt with *epistemology*—the branch of philosophy concerned with the *theory of knowledge*. The aim of Part II was to compare Kuhn and Popper's alternative ideas of the development of scientific knowledge, to the Agile and XP leaders' ideas on the development of software, in order to assess the extent to which Agile software methodologies—and XP in particular—can be considered scientific.

With regard to Popper's ideas in Part II, three main aspects of his epistemology—*philosophy of science*, *evolutionary theory of knowledge*, and *metaphysics*—were used to

provide a philosophical understanding of XP. Firstly, it was shown, using Popper's principle of *falsificationism* as a criterion for demarcating science from non-science, that software engineering, in general, resembles a scientific discipline, due to the susceptibility of software to testing. It was also shown that several of the XP values, principles and practices, as advocated by Beck, are aligned with Popper's principle of falsificationism, and his related ideas of *criticism* and *error elimination*. Secondly, a comparison was made between Popper's four-stage model of the *evolutionary theory of knowledge* and XP's critical *test-driven*, and *incremental* software development approach, in an attempt to illuminate the similarities between a software methodology, and a scientific methodology. It was found that each of the four stages in Popper's model resembles a phase in XP software development, although an additional phase would be required in Popper's model for the analogy to accommodate XP's unique practice of *test-first programming*. This analogy, together with similarities between certain of XP's values and principles, and certain of the principles underlying Popper's four-stage model, showed that Popper's epistemology is useful for understanding the philosophical basis of the small-scale everyday practice of XP software development. Thirdly, by comparing Popper's *three worlds metaphysical model*, which emphasises the superiority of *objective* knowledge, to Polanyi's model of *tacit knowing*, which emphasises *subjective knowledge*, it was shown that Popper's model provides a comprehensive framework for understanding the approach of Agile methodologists towards creating and sharing knowledge, despite their stated reliance on *tacit* or *subjective* knowledge. The discussion of Popper's ideas in Part II concluded with an investigation into the problem of what software actually *is* (an ontological problem), and found Popper's *three worlds metaphysical model* to be useful in providing an understanding of the open and evolving nature of software products, which were conjectured to inhabit Popper's World 3 of objective logical content, alongside scientific theories and artworks.

With regard to Kuhn's ideas in Part II, it was investigated whether the use of his concepts—in particular, "paradigm shift", "scientific revolution", and "crisis"—by several leading figures of the Agile and XP methodologies, especially those affiliated with the Cutter Consortium, could constitute a background theory for Agile software methodologies. The extent to which Kuhn's model of *revolutionary change* in science can meaningfully be transferred to Agile software development, was critically assessed, and it was found that Kuhn's concept of "paradigm shift", when used in its most general sense, provides a prediction

of the large-scale change that would result if an Agile software methodology is chosen to replace completely another software methodology. With regard to XP in particular, it was found that this analogy only coincides with the *full-scale* approach to the adoption of XP, which Beck advocates in the first edition of his *Extreme Programming Explained*, and not to the *incremental* approach, which he advocates in the second edition. A *systematic* comparison between Kuhn's ideas and Agile software development, also revealed that Kuhn's concepts of "paradigm" and "revolution", when applied strictly to Agile methodologies, cannot provide an adequate explanation of the small-scale changes that occur during the day-to-day practice of Agile software development. From these findings, it was conjectured that the Agile methodologists who have used Kuhnian concepts, have done so for *predictive*, rather than *descriptive* purposes. More specifically, Kuhn's concepts may have been used by certain Agile methodologists to predict a fundamental change from traditional to Agile software methodologies rather than to describe the day-to-day practice of Agile software development, in the hope that the Agile software methodologies will bring about a revolution, and trigger a paradigm shift, in software methodology. In relation to XP specifically, it was conjectured that Beck may have gained an insight about the *irrationality of paradigm shifts* from Kuhn, which may have helped him to predict that the rationality of the XP values, principles and practices alone, would not trigger a paradigm shift in software methodology, but, instead, that certain irrational factors would influence the *adoption* of XP. This may explain Beck's citation of Kuhn's *Structure* in the first edition of his book, but it does not seem to explain the same citation in the second edition, since Beck advocates there that the XP methodology be adopted incrementally instead of full-scale. An incremental approach would entail adopting individual XP practices deliberately and rationally for the supposed value they would add to the existing methodology, which suggests there is no irrationality involved in the adoption of XP. The rational and incremental adoption of XP was shown to be accounted for by Popper's *evolutionary* epistemology.

Part III of this dissertation dealt mainly with *ethics*, and aimed to transfer concepts from *social* engineering to *software* engineering, in order to assess both the *democratic* and *authoritarian* aspects of Agile software development and management.

With regard to Popper's ideas in Part III, his notions of *utopian* and *piecemeal* social engineering—which provide a demarcation between *authoritarian* and *democratic* societies

respectively—were used to help understand the divergent approaches towards software development advocated by traditional and Agile software methodologists. Firstly, the holistic approach of traditional software engineering was shown to resemble utopian social engineering, and to share much in common with Kuhn's ideas of *normal science* and paradigm-based research, especially in its inflexibility, and its emphasis on the collective rather than the individual. Secondly, the XP methodologist's approach of *incremental design* was shown to share a *fallibilist idea of rationality* in common with the approach of the *piecemeal social engineer*, and both of these approaches were also shown to reject blueprint planning. Furthermore, the XP team, which upholds the XP values and principles of *courage*, *respect*, *humanity*, *diversity*, and *accepted responsibility*, was favourably depicted in terms of an *open society*. Then it was shown, using as a basis Johnson's critique of the XP methodology, that there are convincing parallels between Kuhn's notion of a "paradigm", and the holistic and irrationalist way in which the XP methodology is *promoted*. Through this investigation, a tension was found between the "closed" versus "open" tendencies in XP software development: on the one hand, XP's *theoretical* values, principles and practices are avowedly democratic—often resembling those found in an open society—whereas, on the other hand, the manner in which the XP methodology is *promoted* and *adopted* appears to be élitist and authoritarian—exhibiting features of a closed society.

Part III also investigated two other important influences on the Agile ethos, namely, the architect, *Christopher Alexander*, and the *Peopleware* authors, *DeMarco and Lister*. It was shown, firstly, that Alexander's approach of *social engineering architecture* resembles Popper's approach of *piecemeal social engineering*, and that both approaches reject full-scale engineering. An important causal link was also found between Popper's ideas and the XP methodology, through Alexander's influence on the Agile and XP leadership. It was conjectured that this link may account partly for the apparent Popperian spirit of several XP values, principles and practices. With regard to *Peopleware*, the authors' ideas regarding *management style* were investigated, and it was found that the "open kimono" style of management advocated by the authors, resembles that of an open society, and that the "defensive" management style, which the authors criticise, have parallels in Popper's critique of *utopian social engineering*. DeMarco and Lister's ideas are important since, as consultants for Agile project management at the Cutter Consortium, they continue to have a significant influence on the various Agile methodologies.

The current prestige and epistemological hegemony of science, technology and "the scientific method" was the motivation behind this dissertation's attempts to investigate the scientific status of software engineering methodologies. Whether it is possible or even desirable for software engineering to imitate the methods of the natural sciences is, however, open to debate. Nonetheless, the findings in this dissertation may be useful—especially in relation to the XP methodology—for the following reasons: firstly, using Popper's principle of *falsificationism* as a criterion for demarcating science from non-science, they show that XP software development resembles a scientific discipline due to its unique emphasis on test-first programming; secondly, they uncover a tension between the apparently *democratic* spirit of the theoretical values, principles and practices, which underlie the XP methodology, and the apparently *authoritarian* manner in which the XP methodology is *propagated* by certain of its leaders; thirdly, they expose the fact that Beck's citation of Kuhn's *Structure* in the second edition of his *Extreme Programming Explained*, unlike in the first edition, seems incongruent with the approach that the second edition advocates, namely, the *incremental adoption* of the XP values, principles and practices; fourthly, by showing that several authoritarian aspects of XP, which Johnson criticises, have parallels in Popper and Fuller's criticisms of Kuhn's philosophy, a means can now be devised to counter these authoritarian tendencies; finally, several aspects of Popper's philosophy, which have not until now been used in relation to Agile methodologies, have been shown to resemble and endorse several aspects of XP software development in particular: in relation to *epistemology*, Popper's principle of *falsificationism* was shown to provide a criterion for understanding the rational and scientific basis of the XP principles and practices, and in relation to *ethics*, Popper's notion of an *open society* was shown to provide an understanding of the avowedly democratic and people-centric approach of XP software development and management.

While the findings in this dissertation do not provide practical guidance for the improvement of the practice of software development per se, they do provide theoretical guidance for choosing rationally between different software methodologies based on criteria—both scientific and ethical—found in Popper's philosophy. The knowledge imparted from this theoretical study can be viewed as an end in itself. The study can be considered a starting point for a philosophical debate and understanding of Agile software methodologies, which will hopefully be extended to other software methodologies in the future. The following is a list of topics for possible future work:

1. The introduction of Popper's ideas of the *open society* into the discourse of software engineering ethics.

2. The development of a software engineering methodology based on Popper's philosophy.

3. The extension of the analogy provided in Appendix F between *open source software* development and Popper's *open society*.

Bibliography

IEEE Standard Glossary of Software Engineering Terminology [online] (1990) [cited April 01, 2008]. Available from: `http://ieeexplore.ieee.org/iel1/2238/4148/00159342.pdf`. (IEE).

Death March Author Ed Yourdon Admits He Was Wrong [online] (1997) [cited April 28, 2008]. Available from: `http://sunsite.uakom.sk/sunworldonline/swol-07-1997/swol-07-bookshelf.html`. (DMA).

Software Engineering: An Unconsummated Marriage [online] (1998) [cited April 29, 2008]. Available from: `http://www.cs.utexas.edu/users/software/1998/parnas-19981208.pdf`. (SEA).

A Process Per Project [online] (1999) [cited April 09, 2008]. Available from: `http://alistair.cockburn.us/index.php/Methodology_per_project`. (PPP).

The New Methodology [online] (2000) [cited April 26, 2008]. Available from: `http://www.martinfowler.com/articles/newMethodology.html`. (TNM).

*TMM*i *Foundation Home* [online] (2002) [cited April 26, 2008]. Available from: `http://www.tmmifoundation.org/`. (TMM).

Project Lifecycles: Waterfall, Rapid Application Development, and All That [online] (2003) [cited April 01, 2008]. Available from: `http://luxworldwide.com/about/whitepapers/waterfall.asp`. (PLW).

Software Development: Agile vs. Disciplined Methods [online] (2003) [cited April 26, 2008]. Available from: `http://www.informit.com/articles/article.aspx?p=101615`. (SDA).

Agile 2008 Conference [online] (2008) [cited April 26, 2008]. Available from: http://www.agile2008.org/. (A2C).

Agile Alliance [online] (2008) [cited April 26, 2008]. Available from: http://www.agilealliance.org/. (AAL).

Burma Report Back [online] (2008) [cited April 18, 2008]. Available from: http://www.avaaz.org/en/burma_report_back/. (BRB).

Christopher Alexander [online] (2008) [cited April 01, 2008]. Available from: http://en.wikipedia.org/wiki/Christopher_Alexander. (CAL).

Chrysler Comprehensive Compensation System [online] (2008) [cited April 28, 2008]. Available from: http://en.wikipedia.org/wiki/Chrysler_Comprehensive_Compensation_System. (CCC).

Cutter Consortium [online] (2008) [cited June 3, 2008]. Available from: http://www.cutter.com. (CCO).

Ed Yourdon [online] (2008) [cited April 28, 2008]. Available from: http://www.yourdon.com/. (EY1).

Ed Yourdon [online] (2008) [cited April 28, 2008]. Available from: http://www.cutter.com/meet-our-experts/eybio.html. (EY2).

Edsger W. Dijkstra [online] (2008) [cited May 17, 2008]. Available from: http://en.wikipedia.org/wiki/Edsger_W._Dijkstra. (EWD).

European Association of Software Science and Technology [online] (2008) [cited April 22, 2008]. Available from: http://www.easst.org/. (EAS).

Explicit Knowledge [online] (2008) [cited May 25, 2008]. Available from: http://en.wikipedia.org/wiki/Explicit_knowledge. (EKN).

Formal Verification [online] (2008) [cited June 8, 2008]. Available from: http://en.wikipedia.org/wiki/Formal_verification. (FVE).

Free and Open Source Software [online] (2008) [cited April 05, 2008]. Available from: http://en.wikipedia.org/wiki/Free_open_source_software. (FOS).

History of Software Engineering [online] (2008) [cited April 29, 2008]. Available from: `http://en.wikipedia.org/wiki/History_of_software_engineering`. (HSE).

Integration Testing [online] (2008) [cited May 10, 2008]. Available from: `http://en.wikipedia.org/wiki/Integration_test`. (ITE).

Is-ought Problem [online] (2008) [cited May 10, 2008]. Available from: `http://en.wikipedia.org/wiki/Is_ought`. (IOP).

Karl Popper Debate [online] (2008) [cited April 06, 2008]. Available from: `http://en.wikipedia.org/wiki/Karl_Popper_debate`. (KPD).

Master Plans [online] (2008) [cited April 01, 2008]. Available from: `http://www.gardenvisit.com/history/theory/garden/landscape/design/articles/landscape/theory/master/plans/vision/concept`. (MPL).

Philosophy of the GNU Project [online] (2008) [cited April 01, 2008]. Available from: `http://www.gnu.org/philosophy/`. (GNU).

Software Crisis [online] (2008) [cited April 28, 2008]. Available from: `http://en.wikipedia.org/wiki/Software_crisis`. (SCR).

Summit 2008 [online] (2008) [cited April 26, 2008]. Available from: `http://www.cutter.com/summit/2008.html`. (S20).

Tacit Knowledge [online] (2008) [cited May 25, 2008]. Available from: `http://en.wikipedia.org/wiki/Tacit_knowledge`. (TKN).

The Free Software Definition [online] (2008) [cited April 01, 2008]. Available from: `http://www.gnu.org/philosophy/free-sw.html`. (FSD).

The Jargon File [online] (2008) [cited April 19, 2008]. Available from: `http://catb.org/jargon/html/`. (TJF).

The Timeless Way of Building [online] (2008) [cited April 01, 2008]. Available from: `http://en.wikipedia.org/wiki/The_Timeless_Way_of_Building`. (TWB).

Unit Testing [online] (2008) [cited May 10, 2008]. Available from: `http://en.wikipedia.org/wiki/Unit_test`. (UTE).

Waterfall Model [online] (2008) [cited April 01, 2008]. Available from: `http://en.wikipedia.org/wiki/Waterfallmodel`. (WMO).

Aichernig, B. (2001). *Systematic Black-Box Testing of Computer-Based Systems through Formal Abstraction Techniques*. Ph.D. thesis, Technische Universität Graz.

Akhgar, B., ed. (2007). *Proceedings of the 15th International Conference on Conceptual Structures (ICCS)*.

Alexander, C. (1964). *Notes on the Synthesis of Form*. Harvard University Press, London.

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press, New York.

Alexander, C. (1999). The Origins of Pattern Theory: The Future of the Theory and the Generation of a Living World. *IEEE Software*, **16**, 71–82.

Alexander, C., Ishikawa, S. & Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York.

Ambler, S. *Survery Says...Agile Has Crossed the Chasm* [online] (2007) [cited April 01, 2008]. Available from: `http://www.ddj.com/architect/200001986`.

Ambler, S. *Agile Software Development: Definition* [online] (2008) [cited April 01, 2008]. Available from: `http://www.agilemodeling.com/essays/agileSoftwareDevelopment.htm`.

Arageorgis, A. & Baltas, A. (1989). Demarcating Technology from Science: Problems and Problem Solving in Technology. *Zeitschrift für Allgemeine Wissenschaftstheorie*, **20**, 212–229.

Asay, M. *Popper: Open Source as Falsification* [online] (2008) [cited April 01, 2008]. Available from: `http://asay.blogspot.com/2005/08/popper-open-source-as-falsification.html`.

Ashby, R. (1964). *Logical Positivism*, 492–508. In O'Connor (1964).

Ayer, A. (1971). *Language, Truth and Logic*. Pelican Books, England.

Bach, J. *Skill Over Process* [online] (2000) [cited April 01, 2008]. Available from: `http://www.cutter.com/content/trends/fulltext/reports/2000/08/index.html`.

Barnard, L. & Botha, R., eds. (2007). *Proceedings of SAICSIT 2007*.

Basden, A. (2008). *Philosophical Frameworks for Understanding Information Systems*. IGI Publishing, USA.

Beck, K. *XP and Culture Change* [online] (2002) [cited April 01, 2008]. Available from: `http://www.cutter.com/research/2002/edge021022.html`.

Beck, K. & Andres, C. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley, London, 1st edn.

Beck, K. & Andres, C. (2005). *Extreme Programming Explained: Embrace Change*. Addison-Wesley, London, 2nd edn.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J. & Thomas, D. *Manifesto for Agile Software Development* [online] (2001a) [cited April 01, 2008]. Available from: `http://www.agilemanifesto.org`.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J. & Thomas, D. *Principles Behind the Agile Manifesto* [online] (2001b) [cited April 13, 2008]. Available from: `http://agilemanifesto.org/principles.html`.

Bishop, J. (1991). Computer Programming: is it Computer Science? *South African Journal of Science*, **87**, 22–33.

Bloomberg, J. *Web Services and the New IT Paradigm* [online] (2002) [cited April 01, 2008]. Available from: `http://www.cutter.com/content/itjournal/fulltext/2002/04/itj0204b.html`.

Boehm, B. (1988). A Spiral Model of Software Development and Enhancement. *Computer*, **21**, 61–72.

Boehm, B. (2002). Get Ready for Agile Methods, with Care. *IEEE Software*, **35**, 64–69.

Bossavit, L. *Popperian Software Design* [online] (2008) [cited June 26, 2008]. Available from: `http://www.bossavit.com/pivot/pivot/entry.php?id=239`.

Brooks, F. (1975). *The Mythical Man-Month*. Addison-Wesley, London, 1st edn.

Brooks, F. (1987). No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, **20**, 10–19.

Bullock, A. & Trombley, S., eds. (1999). *The New Fontana Dictionary of Modern Thought*. Harper Collins Publishers, London, 3rd edn.

Burnstein, I., Suwanassart, T. & Carlson, R. (1996). Developing A Testing Maturity Model For Software Test Process Evaluation and Improvement. *Proceedings of the IEEE International Test Conference on Test and Design Validity*, 581–589.

Bynum, T. *Computer Ethics: Basic Concepts and Historical Overview* [online] (2001) [cited April 01, 2008]. Available from: `http://www.science.uva.nl/~seop/entries/ethics-computer/`.

Carroll, R. (2003). *The Skeptic's Dictionary*. John Wiley & Sons, New Jersey.

Castells, M. (2001). *Informationalism and the Network Society*, 156–178. In Himanen (2001).

Chalmers, A. (1982). *What Is This Thing Called Science?*. Open University Press, England, 2nd edn.

Chau, T., Maurer, F. & Melnik, G. (2003). Knowledge Sharing: Agile Methods vs. Tayloristic Methods. *Proceedings of the Twelfth IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 302–307.

Chromatic. *An Introduction to Extreme Programming* [online] (2001) [cited April 05, 2008]. Available from: `http://www.linuxdevcenter.com/pub/a/linux/2001/05/04/xp_intro.html`.

Coar, K. *The Open Source Definition* [online] (2006) [cited April 04, 2008]. Available from: `http://www.opensource.org/docs/osd`.

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

Cockburn, A. (2002b). Agile Software Development Joins the "Would-be" Crowd. *Cutter IT Journal*, **15**, 6–12.

Cockburn, A. & Highsmith, J. (2001). Agile Software Development: The People Factor. *IEEE Computer*, **34**, 131–133.

Cockburn, A. & Williams, L. (2001). The Costs and Benefits of Pair Programming. *Extreme Programming Examined*, 223–243, Addison-Wesley Longman Publishing Co., Inc.

Constantine, L. (2001). *The Peopleware Papers*. Prentice Hall, New Jersey.

Coutts, D. *The Test Case as a Scientific Experiment* [online] (2008) [cited April 01, 2008]. Available from: `http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=ART&ObjectId=8965`.

Dafermos, G. *Openness and Digital Ethics* [online] (2004) [cited April 01, 2008]. Available from: `http://radio.weblogs.com/0117128/ethical_licensing/Openness_and_Digital_Ethics.html`.

Davies, R. *Agile Paradigm Shift* [online] (2006) [cited April 01, 2008]. Available from: `www.agilenorth.net/DownloadFiles/2006Conference/AgileNorthKeynote.ppt`.

DeMarco, T. & Lister, T. (1999). *Peopleware: Productive Projects and Teams*. Dorset House Publishing Co., New York, 2nd edn.

DeMarco, T., Hruschka, P., Lister, T., McMenamin, S., Robertson, J. & Robertson, S. (2008). *Adrenaline Junkies and Template Zombies: Understanding Patterns of Project Behaviour*. Dorset House Publishing, New York.

DiBona, C., Ackerman, S. & Stone, M. (1999). *Open Sources: Voices from the Open Source Revolution*. O'Reilly, CA.

Dijkstra, E. (1968). Go To Statement Considered Harmful: Letter to the Editor. *Communications of the ACM*, **11**, 147–148.

Dijsktra, E. (1989). On the Cruelty of Really Teaching Computer Science. *Communications of the ACM*, **32**, 1398–1404.

Ehn, P. (1988). *Work-Oriented Development of Software Artifacts*. Arbetslivescentrum, Stockholm.

Feyerabend, P. (1987). *Farewell to Reason*. Verso, London.

Feyerabend, P. (1988). *Against Method*. Verso, London.

Firestone, J. & McElroy, M. *Corporate Epistemology* [online] (2003a) [cited April 01, 2008]. Available from: `www.dkms.com/papers/corporateepistemologyandkm.pdf`.

Firestone, J. & McElroy, M. (2003b). *The Open Enterprise: Building Business Architectures for Openness and Sustainable Innovation*. Hartland Four Corners, VT: KMCI Online Press, `http://www.kmci.org/`.

Flew, A., ed. (1984). *A Dictionary of Philosophy*. The Macmillan Press, London, 3rd edn.

Flor, N. & Hutchins, E. (1991). Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming during Perfective Software Maintenance. In Koenemann-Belliveau *et al.* (1991), 36–64.

Fowler, M. *AppealToAuthority* [online] (2008) [cited April 01, 2008]. Available from: `http://martinfowler.com/bliki/AppealToAuthority.html`.

Fraser, S., Boehm, B., Jr., F.B., DeMarco, T., Lister, T., Rising, L. & Yourdon, E. (2007). Retrospectives on *Peopleware*. In *ICSE COMPANION '07: Companion to the Proceedings of the 29th International Conference on Software Engineering*, 21–24, IEEE Computer Society, Washington, DC, USA.

Fuller, S. (2003). *Kuhn vs Popper: The Struggle for the Soul of Science*. Icon books, UK.

Giguette, R. (2006). Building Objects Out of Plato: Applying Philosophy, Symbolism and Analogy to Software Design. *Communications of the ACM*, **49**, 66–71.

Gotterbarn, D. *Software Engineering Code of Ethics and Professional Practice* [online] (1999) [cited April 01, 2008]. Available from: `http://www.acm.org/about/se-code#full`. Institute of Electrical and Electronics Engineers, Inc., and the Association for Computing Machinery, Inc.

Gotterbarn, D. (2001). Informatics and Professional Responsibility. *Science and Engineering Ethics*, **7**, 221–230.

Gourlay, S. (2002). Tacit Knowledge, Tacit Knowing, or Behaving? *3rd European Organizational Knowledge, Learning, and Capabilities Conference*.

Gruner, S. (2007). The Path to Innovation. *Last Word Column: Innovate Magazine*, **2**, 96.

Hall, W. (2003a). Managing Maintenance Knowledge in the Context of Large Engineering Projects: Theory and Case Study. *Information & Knowledge Management*, **2**, 1–17.

Hall, W. (2003b). Organizational Autopoiesis and Knowledge Management. *ISD '03 Twelfth International Conference on Information Systems Development*, 25–27.

Hall, W. (2005). Biological Nature of Knowledge in the Learning Organization. *The Learning Organization: An International Journal*, **12**, 169–180.

Hamlet, D. *Science, Computer 'Science', Mathematics, and Software Development* [online] (2002) [cited April 01, 2008]. Available from: `http://web.cecs.pdx.edu/~hamlet/QW.pdf`.

Himanen, P. (2001). *The Hacker Ethic and the Spirit of the Information Age*. Secker & Warburg, Great Britain.

Hoare, C. (2003a). Assertions: A Personal Perspective. *IEEE*, **25**, 14–25.

Hoare, C. (2003b). Towards the Verifying Compiler. *LNAI*, **2757**, 151–160.

Jeffries, R. *Extreme Programming and the Capability Maturity Model* [online] (2000) [cited April 01, 2008]. Available from: `http://www.xprogramming.com/xpmag/xp_and_cmm.htm`.

Johnson, D. (2001). *Computer Ethics*. Prentice Hall, New Jersey, 3rd edn.

Johnson, E. *Hacknot: Essays on Software Development* [online] (2006) [cited April 01, 2008]. Available from: `http://www.hacknot.info/hacknot/action/showEntry?eid=73`.

Koenemann-Belliveau, J., Mohen, T. & Robertson, S., eds. (1991). *Empirical Studies of Programmers: Fourth Workshop*.

Krauch, H. (1994). *Systemanalyse. In Handlexikon zur Wissenschaftstheorie*. DTV Wissenschaft, Munich, 2nd edn.

Kruchten, P. (2003). *The Rational Unified Process: An Introduction*. Addison Wesley Professional, Boston, 3rd edn.

Kuhn, T. (1962). *The Structure of Scientific Revolutions*. The University of Chicago Press, Chicago, 1st edn.

Kuhn, T. (1970). *The Structure of Scientific Revolutions*. The University of Chicago Press, Chicago, 2nd edn.

Layman, L., Williams, L. & Cunningham, L. (2004). Exploring Extreme Programming in Context: An Industrial Case Study. 32–41, IEEE Computer Society, Washington, DC, USA, ADC '04: Proceedings of the Agile Development Conference.

Lyytinen, K. & Klein, H. (1985). *The Critical Theory of Juergen Habermas as a Basis for a Theory of Information Systems*, 219–236. Elsevier Science Publishers, Amsterdam, research Methods in Information Systems.

Magee, B. (1973). *Popper*. Fontana Press, Glasgow.

Magee, B., ed. (1986). *Modern British Philosophy*. Oxford University Press, Oxford.

Mah, M. *Defending the Paradigm (or Designing the Future)* [online] (2004) [cited April 01, 2008]. Available from: `http://www.cutter.com/content/trends/fulltext/advisor/2004/btt040226.html`.

Makins, M., Isaacs, A. & Adams, D., eds. (1991). *Collins English Dictionary*. Harper Collins Publishers, Glasgow, 3rd edn.

Margolis, J. (1980). *Art and Philosophy*. The Harvester Press, Sussex.

Marick, B. (2004). Methodology Work is Ontology Work. *ACM Sigplan Notices*, **39**, 64–72.

Marzolf, T. & Guttman, M. *Systems Minus Systems Thinking Equals Big Trouble* [online] (2002) [cited April 01, 2008]. Available from: `http://www.cutter.com/content/alignment/fulltext/summaries/2002/11/index.html`.

McDowell, C., Werner, L., Bullock, H. & Fernald, J. (2002). The Effects of Pair-programming on Performance in an Introductory Programming Course. *SIGCSE Bull.*, **34**, 38–42.

McElroy, M. *"Deep" Knowledge Management* [online] (2002) [cited April 01, 2008]. Available from: `http://www.macroinnovation.com/images/DeepKMbyM.W.McElroy.pdf`.

Meyer, S. *Pragmatic Versus Structured Computer Programming* [online] (2008) [cited April 01, 2008]. Available from: `http://www.pragmatic-c.com/docs/structprog.pdf`.

Miller, D., ed. (1983). *A Pocket Popper*. Fontana Press, Glasgow.

Moor, J. (1985). What is Computer Ethics? *Metaphilosophy*, **16**, 266–275.

Moroz, T. *Open Source and Open Society: Using Plone To Build Community Online* [online] (2008) [cited April 01, 2008]. Available from: `http://maurits.vanrees.org/weblog/archive/2007/10/open-source-and-open-society`.

Moss, M. (2003). Why Management Theory Needs Popper: The Relevance of Falsification. *Philosophy of Management*, **3**, 31–42.

Naur, P. & Randell, B. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee* [online] (1969) [cited April 22, 2008]. Available from: `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/`.

Nonaka, I. & Takeuchi, H. (1995). *The Knowledge-Creating Company*. Oxford University Press, Oxford.

Northover, M., Boake, A. & Kourie, D. (2006). *Karl Popper's Critical Rationalism in Agile Software Development*. In Scharfe *et al.* (2006), 360–373.

Northover, M., Northover, A., Gruner, S., Kourie, D. & Boake, A. (2007a). *Agile Software Development: A Contemporary Philosophical Perspective*. In Barnard & Botha (2007), 106–115.

Northover, M., Northover, A., Gruner, S., Kourie, D. & Boake, A. (2007b). *Extreme Programming: A Kuhnian Revolution?* In Akhgar (2007), 199–204.

Northover, M., Northover, A., Gruner, S., Kourie, D. & Boake, A. (2008). Towards a Philosophy of Software Development. *Journal for General Philosophy of Science*, **39**.

Norton, D. & Murphy, T. *Agile Essence: Extreme Programming* [online] (2007) [cited April 01, 2008]. Available from: `http://0-www.gartner.com.innopac.up.ac.za/resources/150600/150685/agile_essence_extreme_progra_150685.pdf`. Gartner, Inc. and/or its Affiliates. ID Number: G00150685.

O'Connor, D. (1964). *A Critical History of Western Philosophy*. The Free Press of Glencoe, New York.

Pauleen, D. (2007). *Cross-cultural Perspectives on Knowledge Management*. Libraries Unlimited, London.

Paulk, M. (2001). Extreme Programming from a CMM Perspective. *IEEE Software*, **18**, 19–26.

Penrose, R. (1989). *The Emperor's New Mind*. Oxford University Press, USA Inc.

Polanyi, M. (1967). *The Tacit Dimension*. Doubleday, New York.

Popper, K. (1959). *The Logic of Scientific Discovery*. Routledge, London.

Popper, K. (1963). *Conjectures and Refutations: The Growth of Scientific Knowledge*. Routledge, London.

Popper, K. (1966). *The Open Society and Its Enemies: Volume 1: The Spell of Plato*. Routledge, London.

Popper, K. (1999). *All Life is Problem Solving*. Routledge, London.

Puccinelli, B., Boyd, S. & Kull, M. *Knowledge Management: Exploiting Your Greatest Resource* [online] (2000) [cited April 01, 2008]. Available from: `http://www.cutter.com/content/alignment/fulltext/reports/2000/08/index.html`.

Quinton, A. (1964). *Contemporary British Philosophy*, 531–556. In O'Connor (1964).

Raymond, E. (1999). *The Cathedral & the Bazaar*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Rayside, D. & Campbell, G. (2000). An Aristotelian Understanding of Object-Oriented Programming. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 337–353, ACM, New York, NY, USA.

Royce, W. (1970). Managing the Development of Large Software Systems: Concepts and Techniques. *Proceedings of IEEE WESCON*, 1–9.

Russell, B. (1961). *History of Western Philosophy*. George Allen & Unwin Ltd, London, 2nd edn.

Scharfe, H., Hitzler, P. & Ohrstrom, P., eds. (2006). *Proceedings of the 14th International Conference on Conceptual Structures (ICCS)*.

Schwaber, K. *The Agile Alliance Revolution* [online] (2001) [cited April 01, 2008]. Available from: `http://www.cutter.com/content/project/fulltext/updates/2001/epmu0108.html`.

Shaw, M. (1996). Three Patterns That Help Explain the Development of Software Engineering. *Position paper for Dagstuhl Seminar on Software Architecture*.

Smith, R. (1997). The Historical Roots of Concurrent Engineering Fundamentals. *IEEE Transactions on Engineering Management*, **44**, 67–78.

Snelting, G. (1997). Paul Feyerabend und die Softwaretechnologie. *Softwaretechnik-Trends*, **17**, 55–59.

Soros, G. (2006). *The Age of Fallibility: The Consequences on the War of Terror*. Phoenix, London.

Stalder, F. *Open Source as a Social Principle* [online] (2003) [cited April 01, 2008]. Available from: `http://felix.openflows.com/html/os_social_principle_ns.html`.

Theunissen, W., Boake, A. & Kourie, D. (2005). In Search of the Sweet Spot: Agile Open Collaborative Corporate Software Development. *Proceedings of SAICSIT 2005*, 268–277, White River, South Africa.

Totten, W. *Software Reliability: A Development-style Approach* [online] (1999) [cited April 01, 2008]. Available from: `http://glaciated.org/reliability/`.

Weinberg, G.M. (1971). *The Psychology of Computer Programming*. Van Nostrand Reinhold Company, New York.

Welsh, T. *Is the Market Right for MDA?* [online] (2001) [cited April 01, 2008]. Available from: `http://www.cutter.com/content/architecture/fulltext/updates/2004/eau0415.html`.

Wernick, P. & Hall, T. (2004). Can Thomas Kuhn's Paradigms Help Us Understand Software Engineering? *European Journal of Information Systems*, **13**, 235–243.

Williams, L., Kessler, R., Cunningham, W. & Jeffries, R. (2000). Strengthening the Case for Pair-programming. *IEEE Software*, 19–25, July/August.

Williams, L., Krebs, W., Layman, L. & Anton, A. (2004). Toward a Framework for Evaluating Extreme Programming. *8th International Conference on Empirical Assessment in Software Engineering (EASE 04)*, 11–20.

Windholtz, M. *XP Universe 2002* [online] (2002) [cited April 01, 2008]. Available from: `www.objectwind.com/papers/XPUniverse2002.pdf`.

Wittgenstein, L. (1958). *Philosophical Investigations*. Basil Blackwell Ltd, Oxford.

Yourdon, E. *Microsoft Criticises the Open-source Concept* [online] (2001a) [cited April 01, 2008]. Available from: `http://www.cutter.com/content/trends/fulltext/advisor/2001/btt010517.html`.

Yourdon, E. *Paradigm Shifts* [online] (2001b) [cited April 01, 2008]. Available from: `http://www.cutter.com/content/trends/fulltext/advisor/2001/btt011115.html`.

Yourdon, E. *Peer-to-peer Computing: The Next Big Paradigm Shift?* [online] (2001c) [cited April 01, 2008]. Available from: `http://www.cutter.com/content/trends/fulltext/advisor/2001/btt010614.html`.

Yourdon, E. *The XP Paradigm Shift* [online] (2001d) [cited April 01, 2008]. Available from: `http://www.cutter.com/content/trends/fulltext/advisor/2001/btt010510.html`.

Yourdon, E. (2007). Celebrating Peopleware's 20^{th} Anniversary. *IEEE Computer Society*, **24**, 96–100, 2007 International Conference on Software Engineering.

Appendix A

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

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

That is, while there is value in the items on
the right, we value the items on the left more.[1]

[1]Beck *et al.* (2001a)

Principles Behind the Agile Manifesto

We follow these principles:

> Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
>
> Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
>
> Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
>
> Business people and developers must work together daily throughout the project.
>
> Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
>
> The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
>
> Working software is the primary measure of progress.
>
> Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
>
> Continuous attention to technical excellence and good design enhances agility.
>
> Simplicity—the art of maximizing the amount of work not done—is essential.
>
> The best architectures, requirements, and designs emerge from self-organizing teams.

> At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.[2]

[2]Beck *et al.* (2001b)

Appendix B

The XP Values, Principles and Practices

Values

- Communication
- Simplicity
- Feedback
- Courage
- Respect

Principles

- Humanity
- Economics
- Mutual Benefit
- Self-similarity
- Improvement
- Diversity
- Reflection
- Flow
- Opportunity
- Redundancy
- Failure
- Quality
- Baby steps
- Accepted Responsibility

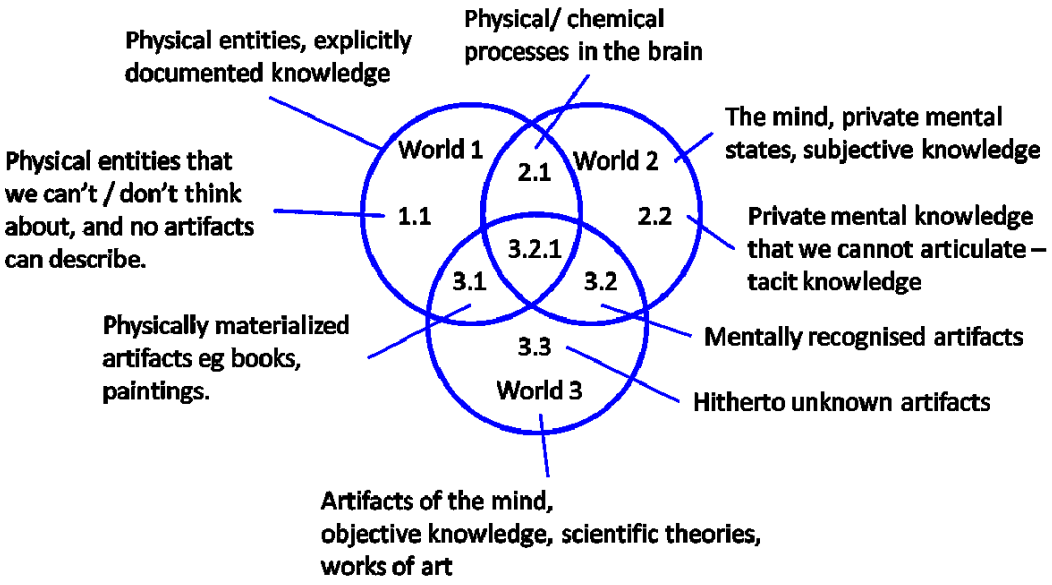
Practices

- Sit Together
- Whole Team
- Informative Workspace
- Energised Work
- Pair Programming
- Stories
- Weekly Cycle
- Slack
- Continuous Integration
- Test-first Programming
- Incremental Design
- Real Customer Involvement
- Incremental Deployment
- Team Continuity
- Shrinking Teams
- Root-cause Analysis
- Shared Code
- Code and Tests
- Single Code Base
- Daily Deployment
- Negotiated Scope Contract
- Pay-per-use[a]

[a] Beck & Andres (2005)

Appendix C

Popper's Three Worlds Metaphysical Model



3.2.1. The thinking subject recording the objective logical content of his/her thoughts in a physical form/medium.

Figure C.1: Popper's Three Worlds Metaphysical Model

Appendix D

History of Computer Ethics

This appendix provides a historical account of *computer ethics*, for Part III of this dissertation.

Perhaps surprisingly, the history of computer ethics spans for approximately as long as the history of software engineering. However, unlike software engineering, computer ethics gained recognition as an independent field only in the 1990s. Some consider computer ethics to be a branch of *practical philosophy*, others a branch of *applied ethics*. According to Bynum (2001), there are also different interpretations of the term "computer ethics", both narrow and broad. The *narrow* interpretation considers the application of traditional ethical theories like utilitarianism and Kantianism to the use of computer technology, while the *broader* interpretation considers standards of professional practice, *codes of conduct*, and the like. Consequently, there are a wide variety of definitions of the term. For the purposes of this background of the history of computer ethics, the following definition will suffice:

> Computer ethics is the analysis of the nature and social impact of computer technology and the corresponding formulation and justification of policies for the ethical use of such technology. (Moor, 1985)

1940s and 1950s

Computer ethics has its roots in the work of *Norbert Wiener* who, during World War II, worked on automatic aiming and firing of anti-aircraft guns. This work not only helped Wiener to formalise the field now known as *cybernetics*, but also led him to question the

ethical implications of his work. "He perceptively foresaw revolutionary social and ethical consequences" (Bynum, 2001). Although Wiener did not explicitly use the term "computer ethics", his work certainly laid the early foundations for the field, providing a sound basis for the ensuing research on computer ethics.

1960s

The prominent figure of the sixties, in the computer ethics field, was undoubtedly *Donn Parker*, who investigated computer crime and other unethical computer technology activities. Parker led the development of the first *Code of Professional Conduct* for the ACM, thereby giving the field increasing momentum and importance.

1970s

In the early 1970s, *Joseph Weizenbaum* wrote a computer program called *ELIZA*—a crude imitation of a Rogerian psychotherapist. The computer program astounded many, and some even thought it proof that computers would eventually automate many human tasks, including psychotherapy.

In the mid 1970s, the term "computer ethics" was first coined by *Walter Maner*. He was highly influential in the teaching of computer ethics in the USA, especially at the university-level. Maner used the *utilitarian* ethics of Bentham and Mill, as well as the *rationalist* ethics of Kant, in his work. He also popularised the notion of an ethics "Starter Kit" containing curriculum materials and other pedagogical material for students.

1980s

In the mid 1980s, James Moor published his influential article "What Is Computer Ethics?" (Moor, 1985). In it he argued that computer ethics is justified as a discipline in its own right (between science and ethics) because computers provide new capabilities, which, in turn, create new choices for action, resulting in newly emerging values. Therefore, the central task of computer ethics, Moor argues, is to create policies to guide actions under these new circumstances. These policies include both personal and social policies for the ethical use of computer technology.

Moor provides two arguments for the importance of the computer ethics discipline. The first involves the claim that computers are truly revolutionary due to their *logical*

malleability, which means "they can be shaped and molded to do any activity that can be characterized in terms of inputs, outputs, and connecting logical operations". Moor's argument is as follows:

> Logical malleability assures the enormous application of computer technology. This will bring about the Computer Revolution. During the Computer Revolution many of our human activities and social institutions will be transformed. These transformations will leave us with policy and conceptual vacuums about how to use computer technology. Such policy and conceptual vacuums are the marks of basic problems within computer ethics. Therefore, computer ethics is a field of substantial practical importance.

Moor's second argument is not based on the assumption of a computer revolution. Instead, the argument centers on the observation of the *invisibility* of computer operations. This invisibility manifests itself in *invisible abuse*—the spread of computer viruses with the intent to do harm, for example—in *invisible programming values*—the implicit assumptions developers make about the use of the software, which impact on the design of the software, for example—and in *invisible complex calculations*—calculations too enormous for humans to oversee. This invisibility, while often beneficial, makes human beings vulnerable, and means that policies need to be determined that help human beings decide when to trust computers and when not to.

Moor's definition of the term "computer ethics", which includes the concept of policy and conceptual vacuums, as detailed above, is widely considered the most influential definition in the field today even though it is not grounded in any specific professional philosophy.

Also during the 1980s, Deborah Johnson published the first textbook, *Computer Ethics* (Johnson, 2001), which was considered to be the defining textbook in the field for more than a decade. Similarly to Maner, Johnson used both utilitarianism and Kantianism in her work. However, unlike Maner and Moor, she did not believe that computers create wholly new moral problems and, in fact, later stated that she believed computer ethics as a discipline would one day be absorbed into traditional ethics.

1990s

The 1990s saw the first international multidisciplinary conference on computer ethics held, which was considered by many to be a major milestone in the field. Arguably, this brought

about a "second generation" of computer ethics, which spawned a whole set of new university courses, conferences, journals, and textbooks in the field.

Later, *Donald Gotterbarn* highlighted the importance of computer ethics by stating that:

> Computer professionals have specialized knowledge and often have positions with authority and respect in the community. For this reason, they are able to have a significant impact upon the world, including many of the things that people value. Along with such power to change the world comes the duty to exercise that power responsibly. (Gotterbarn, 2001)

Appendix E

The Software Engineering Code of Ethics and Professional Practice

Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment, and society at large. Software engineers are those who contribute, by direct participation or by teaching, to the analysis, specification, design, development, certification, maintenance, and testing of software systems. Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession. In accordance with that commitment, software engineers shall adhere to the following Code of Ethics and Professional Practice.

The Code contains eight Principles related to the behavior of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors, and policy makers, as well as trainees and students of the profession. The Principles identify the ethically responsible relationships in which individuals, groups, and organizations participate and the primary obligations within these relationships. The Clauses of each Principle are illustrations of some of the obligations included in these relationships. These obligations are founded in the software engineer's humanity, in special care owed to people affected by the work of software engineers, and in the unique elements of the practice of software engineering. The Code prescribes these as obligations of anyone claiming to be or

aspiring to be a software engineer.

It is not intended that the individual parts of the Code be used in isolation to justify errors of omission or commission. The list of Principles and Clauses is not exhaustive. The Clauses should not be read as separating the acceptable from the unacceptable in professional conduct in all practical situations. The Code is not a simple ethical algorithm that generates ethical decisions. In some situations, standards may be in tension with each other or with standards from other sources. These situations require the software engineer to use ethical judgment to act in a manner that is most consistent with the spirit of the Code of Ethics and Professional Practice, given the circumstances.

Ethical tensions can best be addressed by thoughtful consideration of fundamental principles, rather than blind reliance on detailed regulations. These Principles should influence software engineers to consider broadly who is affected by their work; to examine if they and their colleagues are treating other human beings with due respect; to consider how the public, if reasonably well informed, would view their decisions; to analyze how the least empowered will be affected by their decisions; and to consider whether their acts would be judged worthy of the ideal professional working as a software engineer. In all these judgments concern for the health, safety and welfare of the public is primary; that is, the "Public Interest" is central to this Code.

The dynamic and demanding context of software engineering requires a code that is adaptable and relevant to new situations as they occur. However, even in this generality, the Code provides support for software engineers and managers of software engineers who need to take positive action in a specific case by documenting the ethical stance of the profession. The Code provides an ethical foundation to which individuals within teams and the team as a whole can appeal. The Code helps to define those actions that are ethically improper to request of a software engineer or teams of software engineers.

The Code is not simply for adjudicating the nature of questionable acts; it also has an important educational function. As this Code expresses the consensus of the profession on ethical issues, it is a means to educate both the public and aspiring professionals about the ethical obligations of all software engineers.

PRINCIPLES

Principle 1: Public

Software engineers shall act consistently with the public interest. In particular, software engineers shall, as appropriate:

1. Accept full responsibility for their own work.
2. Moderate the interests of the software engineer, the employer, the client, and the users with the public good.
3. Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy, or harm the environment. The ultimate effect of the work should be to the public good.
4. Disclose to appropriate persons or authorities any actual or potential danger to the user, the public, or the environment, that they reasonably believe to be associated with software or related documents.
5. Cooperate in efforts to address matters of grave public concern caused by software, its installation, maintenance, support, or documentation.
6. Be fair and avoid deception in all statements, particularly public ones, concerning software or related documents, methods, and tools.
7. Consider issues of physical disabilities, allocation of resources, economic disadvantage, and other factors that can diminish access to the benefits of software.
8. Be encouraged to volunteer professional skills to good causes and to contribute to public education concerning the discipline.

Principle 2: Client and employer

Software engineers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest. In particular, software engineers shall, as appropriate:

1. Provide service in their areas of competence, being honest and forthright about any limitations of their experience and education.

2. Not knowingly use software that is obtained or retained either illegally or unethically.

3. Use the property of a client or employer only in ways properly authorized, and with the client's or employer's knowledge and consent.

4. Ensure that any document upon which they rely has been approved, when required, by someone authorized to approve it.

5. Keep private any confidential information gained in their professional work, where such confidentiality is consistent with the public interest and consistent with the law.

6. Identify, document, collect evidence, and report to the client or the employer promptly if, in their opinion, a project is likely to fail, to prove too expensive, to violate intellectual property law, or otherwise to be problematic.

7. Identify, document, and report significant issues of social concern, of which they are aware, in software or related documents, to the employer or the client.

8. Accept no outside work detrimental to the work they perform for their primary employer.

9. Promote no interest adverse to their employer or client, unless a higher ethical concern is being compromised; in that case, inform the employer or another appropriate authority of the ethical concern.

Principle 3: Product

Software engineers shall ensure that their products and related modifications meet the highest professional standards possible. In particular, software engineers shall, as appropriate:

1. Strive for high quality, acceptable cost, and a reasonable schedule, ensuring significant trade-offs are clear to and accepted by the employer and the client, and are available for consideration by the user and the public.

2. Ensure proper and achievable goals and objectives for any project on which they work or propose.

3. Identify, define, and address ethical, economic, cultural, legal, and environmental issues related to work projects.

4. Ensure that they are qualified for any project on which they work or propose to work, by an appropriate combination of education, training, and experience.

5. Ensure that an appropriate method is used for any project on which they work or propose to work.

6. Work to follow professional standards, when available, that are most appropriate for the task at hand, departing from these only when ethically or technically justified.

7. Strive to fully understand the specifications for software on which they work.

8. Ensure that specifications for software on which they work have been well documented, satisfy the user's requirements, and have the appropriate approvals.

9. Ensure realistic quantitative estimates of cost, scheduling, personnel, quality, and outcomes on any project on which they work or propose to work and provide an uncertainty assessment of these estimates.

10. Ensure adequate testing, debugging, and review of software and related documents on which they work.

11. Ensure adequate documentation, including significant problems discovered and solutions adopted, for any project on which they work.

12. Work to develop software and related documents that respect the privacy of those who will be affected by that software.

13. Be careful to use only accurate data derived by ethical and lawful means, and use it only in ways properly authorized.

14. Maintain the integrity of data, being sensitive to outdated or flawed occurrences.

15. Treat all forms of software maintenance with the same professionalism as new development.

Principle 4: Judgment

Software engineers shall maintain integrity and independence in their professional judgment. In particular, software engineers shall, as appropriate:

1. Temper all technical judgments by the need to support and maintain human values.

2. Only endorse documents either prepared under their supervision or within their areas of competence and with which they are in agreement.

3. Maintain professional objectivity with respect to any software or related documents they are asked to evaluate.

4. Not engage in deceptive financial practices such as bribery, double billing, or other improper financial practices.

5. Disclose to all concerned parties those conflicts of interest that cannot reasonably be avoided or escaped.

6. Refuse to participate, as members or advisors, in a private, governmental, or professional body concerned with software-related issues in which they, their employers, or their clients have undisclosed potential conflicts of interest.

Principle 5: Management

Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance. In particular, those managing or leading software engineers shall, as appropriate:

1. Ensure good management for any project on which they work, including effective procedures for promotion of quality and reduction of risk.

2. Ensure that software engineers are informed of standards before being held to them.

3. Ensure that software engineers know the employer's policies and procedures for protecting passwords, files, and information that is confidential to the employer or confidential to others.

4. Assign work only after taking into account appropriate contributions of education and experience tempered with a desire to further that education and experience.

5. Ensure realistic quantitative estimates of cost, scheduling, personnel, quality, and outcomes on any project on which they work or propose to work, and provide an uncertainty assessment of these estimates.

6. Attract potential software engineers only by full and accurate description of the conditions of employment.

7. Offer fair and just remuneration.
8. Not unjustly prevent someone from taking a position for which that person is suitably qualified.
9. Ensure that there is a fair agreement concerning ownership of any software, processes, research, writing, or other intellectual property to which a software engineer has contributed.
10. Provide for due process in hearing charges of violation of an employer's policy or of this Code.
11. Not ask a software engineer to do anything inconsistent with this Code.
12. Not punish anyone for expressing ethical concerns about a project.

Principle 6: Profession

Software engineers shall advance the integrity and reputation of the profession consistent with the public interest. In particular, software engineers shall, as appropriate:

1. Help develop an organizational environment favorable to acting ethically.
2. Promote public knowledge of software engineering.
3. Extend software engineering knowledge by appropriate participation in professional organizations, meetings, and publications.
4. Support, as members of a profession, other software engineers striving to follow this Code.
5. Not promote their own interest at the expense of the profession, client, or employer.
6. Obey all laws governing their work, unless, in exceptional circumstances, such compliance is inconsistent with the public interest.
7. Be accurate in stating the characteristics of software on which they work, avoiding not only false claims but also claims that might reasonably be supposed to be speculative, vacuous, deceptive, misleading, or doubtful.
8. Take responsibility for detecting, correcting, and reporting errors in software and associated documents on which they work.

9. Ensure that clients, employers, and supervisors know of the software engineer's commitment to this Code of Ethics, and the subsequent ramifications of such commitment.

10. Avoid associations with businesses and organizations which are in conflict with this Code.

11. Recognize that violations of this Code are inconsistent with being a professional software engineer.

12. Express concerns to the people involved when significant violations of this Code are detected unless this is impossible, counterproductive, or dangerous.

13. Report significant violations of this Code to appropriate authorities when it is clear that consultation with people involved in these significant violations is impossible, counterproductive, or dangerous.

Principle 7: Colleagues

Software engineers shall be fair to and supportive of their colleagues. In particular, software engineers shall, as appropriate:

1. Encourage colleagues to adhere to this Code.

2. Assist colleagues in professional development.

3. Credit fully the work of others and refrain from taking undue credit.

4. Review the work of others in an objective, candid, and properly documented way.

5. Give a fair hearing to the opinions, concerns, or complaints of a colleague.

6. Assist colleagues in being fully aware of current standard work practices including policies and procedures for protecting passwords, files, and other confidential information, and security measures in general.

7. Not unfairly intervene in the career of any colleague; however, concern for the employer, the client, or public interest may compel software engineers, in good faith, to question the competence of a colleague.

8. In situations outside of their own areas of competence, call upon the opinions of other professionals who have competence in those areas.

Principle 8: Self

Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession. In particular, software engineers shall continually endeavor to:

1. Further their knowledge of developments in the analysis, specification, design, development, maintenance, and testing of software and related documents, together with the management of the development process.

2. Improve their ability to create safe, reliable, and useful quality software at reasonable cost and within a reasonable time.

3. Improve their ability to produce accurate, informative, and well-written documentation.

4. Improve their understanding of the software and related documents on which they work and of the environment in which they will be used.

5. Improve their knowledge of relevant standards and the law governing the software and related documents on which they work.

6. Improve their knowledge of this Code, its interpretation, and its application to their work.

7. Not give unfair treatment to anyone because of any irrelevant prejudices.

8. Not influence others to undertake any action that involves a breach of this Code.

9. Recognize that personal violations of this Code are inconsistent with being a professional software engineer.[1]

[1]Gotterbarn (1999)

Appendix F

Free Open Source Software and Popper's Open Society

The aim of the discussion in this appendix is to provide an account of the similarities between the values underlying Popper's notion of an *open society*, and those underlying *Free Open Source Software Development (FOSSD)*.

Free Open Source Software (FOSS) is "software which is liberally licensed to grant the right of users to study, change, and improve its design through the availability of its source code" (FOS, 2008). FOSS is opposed to *proprietary* or copyrighted software, which is bound by restrictive commercial licensing, and whose source code is unavailable to the public. One of the most significant examples to date of these two contrasting models of software development, is the open source *Linux* operating system versus the closed source *Windows* operating system:

> Anyone can download Linux for free, but this is not the primary difference between Linux and Windows. What distinguishes Linux from the dominant commercial software model epitomized by Microsoft's products is first and foremost its openness: in the same way that scientific researchers allow all others in their fields to examine and use their findings, to be tested and developed further, hackers[1] who take part in the Linux project permit all others to use, test, and

[1]Hackers are defined as "people who program enthusiastically and who believe that information-sharing is a powerful positive good, and that it is their ethical duty to share their expertise by writing free software and facilitating access to information and to computing resources wherever possible" (TJF, 2008). It is important to note that the meaning of the term "hacker" changed later, and is still often used today, instead of the term "cracker", to describe computer criminals like virus writers. This section uses "hacker" in the

> develop their programs. In research, this is known as the scientific ethic. In the field of computer programming, it is called the open-source model.[2]

The relation between FOSSD—a process that a large network of volunteer programmers follow to develop and maintain OSS over the Internet in view of the public—and Agile Software Development (ASD) (the specific method of software development this dissertation considers), is often contested. Some software engineers argue that FOSSD is just another Agile software development approach, whereas others claim that the FOSSD approach is fundamentally incompatible with ASD. Still others suggest that the benefits of both approaches can be gained by adopting a hybrid of the two approaches (Theunissen *et al.*, 2005). With regard to XP in particular, Chromatic (2001) argues:

> Nothing in XP prevents source code from being given away freely. Nothing in open source development precludes a feature-at-a-time development style. In many ways, [open source and the XP methodology] complement each other—user-driven focus, continual code review, short release cycle, and tight feedback loops. There's more in common than one might expect at first glance.

While both FOSSD and ASD share values in common (such as openness and flexibility in the face of change), which are similar to those underlying Popper's notion of an *open society*, the aim of this appendix is not to investigate these commonalities in detail. Rather the intention here is twofold. Firstly, this section will investigate how far the *hacker ethic*, which underlies the FOSSD approach, coincides with the values of an open society, as advocated by Popper. Secondly, it will investigate whether the FOSSD methodology resembles Popper's scientific method.

Before investigating these two aspects, the following should be noted: FOSS encompasses both Open Source Software (OSS) and Free Software (FS). The Open Source Initiative (OSI) defines[3] OSS by emphasising that users may make use of, modify, and improve software without restriction, and may redistribute the software with or without modification. The Free Software Foundation (FSF) emphasise that FS, in addition to being open

sense of the original definition. Hackers believe that system-hacking for fun and exploration is ethically acceptable as long as the hacker commits no theft, vandalism, or breach of confidentiality. In fact, the term 'ethical hacker' or 'white hat' hacker , in contrast to 'black' hat hacker, was coined to represent software developers who focus on securing system.

[2] Himanen (2001) (p. 180)

[3] A complete listing of the *Open Source Definition* is included in Appendix G.

to the public, is defined by the *freedom* users have to run the software, study how it works, release improvements back into the community, and redistribute copies to other members of the community[4]. These two ideologically distinct movements in software engineering—the FS movement (1983) and the OSS movement (1998)—place different emphases on the meaning of openness. The FSF stresses that openness is an aspect of (digital) freedom, and thus the FSF's rhetoric is more politically flavoured. It emphasises the value that openness adds to society at large and points out that the word "free" refers to a matter of liberty, rather than to price. The OSI, while not dismissing the social value of openness, emphasises that the value lies predominantly in producing higher quality software. Therefore, it seems that FS is more relevant than OSS to the topic of discussion in this appendix, since it emphasises values like "freedom" and "liberty" more prominently. Nonetheless, this section will use the generic term "Free/Open Source Software (FOSS)", which was created to describe open source software, which is also free.

From the Scientific Ethic to the Hacker Ethic

FOSS is said to have stemmed from the *hacker ethic* whose historical precedent, according to Himanen (the author of *The Hacker Ethic* (Himanen, 2001) who was introduced in section 3), is in the academic or *scientific ethic*. One of the cornerstones of the scientific ethic lies in the idea that scientific knowledge must be public. Himanen argues that many hackers, in accordance with this ethic, distribute the results of their creativity openly, for others to use, test, and develop further. This section will investigate whether the hackers' method of producing software and releasing it—along with its source code—into the community, resembles Popper's model of the *evolutionary theory of knowledge*. In addition, it will investigate whether aspects of the OSSD approach have parallels in Popper's *three worlds metaphysical* model.

In order to investigate the similarities between Popper's model, which was discussed earlier in section 5.2, and the open-source model, it will first be necessary to provide a more detailed description of the latter model. According to Himanen:

> Generally speaking, this open-source model can be described as follows: it all begins with a problem or goal someone finds personally significant. That person may release just the problem or goal itself, but usually he or she will also provide

[4]Appendix H provides a complete description of the *Free Software Definition*, created by Stallman.

> a Solution—version 0.1.1, to use the Linux numbering system. In the open model, a recipient has the right to freely use, test, and develop the Solution. This is possible only if the information that has led to the Solution (the source) has been passed on with it. In the open-source model, the release of these rights entails two obligations: these same rights have to be passed on when the original Solution or its refined version (0.1.2) is shared, and the contributors must always be credited whenever either version is shared. All this is a shared process, in which the participants move gradually—or sometimes even by leaps and bounds (say, a shift from version 0.y.z to version 1.y.z)—to better versions. In practice, of course, projects follow this idealized model to a greater or lesser extent.[5]

The similarities between the open source model and Popper's model of the *evolutionary theory of knowledge* (as well as XP's *incremental design* approach), are immediately evident from Himanen's description. From section 5.2, the reader will recall that Popper emphasises that all knowledge begins with problem-solving. Indeed, the point of departure in Popper's four-stage model is a problem (P_1) identified by the scientist, which can be considered analogous with the "problem or goal" in the open-source model. The second stage of Popper's model involves forming a bold hypothesis as a tentative solution *TS* to P_1. *TS* can be considered analogous with the "Solution" released by the hacker into the broader community. The acknowledgment in the release notes of the use of another OSS engineer's source code as part of one's own source code, also has parallels in the academic and scientific ethic, whereby researchers are obliged to acknowledge their use of other researchers' ideas in their bibliographies. During the third stage of Popper's model, error elimination (*EE*), scientists attempt to refute the solution by independently subjecting it to severe testing and scrutiny. A similar critical process is evident in the open-source model since the "Solution", containing open source code, is released into the broader software community, where it can be publicly scrutinised. In fact, it can be argued that the "Solution" in the open source model is subjected to even more harsh scrutiny than in the scientific model, since it is released into a wider community via the Internet, and is open not only to software engineers, but also to the general public. Openness is clearly crucial for receiving

[5] Himanen (2001) (pp. 67-68)

this type of feedback from the community. The benefits of this distributed peer review process is emphasised by *Linus's Law*:[6] "Given enough eyeballs all bugs are shallow", which highlights that the more widely available the source code is for public testing, scrutiny, and experimentation, the more rapidly errors will be discovered and eliminated. The similarities between this approach and Popper's emphasis on rigorous testing, as well as his principle of *falsificationism*, are evident. Open source code (and the test cases that show the problem being solved) also ensures that a solution to a problem is repeatable, a requirement that is essential to any scientific discipline. The emphasis on criticism in OSSD encourages *competition*—another important principle in scientific disciplines—resulting in competing software products within the FOSS community. The final stage of Popper's model, P_2, results in a solution with new problems, which can be considered analogous with "version (0.1.2)" in Himanen's description. Finally, the cycles of Popper's model resemble the release of incremental improvements to the "Solution" over time, although, in the open-source model, the improvements are often in the form of additional features, whereas, in Popper's model, improvements would result in a more refined understanding of reality.

In relation to these improvements, Himanen argues that the (evolutionary) model (as described above), "continuously provides new additions ('development versions') to what has already been achieved", and that:

> Much more rarely, there is an entire 'paradigm shift', to use the expression that the philosopher of science Thomas Kuhn introduced in his book *The Structure of Scientific Revolutions*.

Even though Himanen believes an evolutionary model is more common than a revolutionary model with regard to improvements to FOSS, his reference to Kuhn's 'paradigm shift' can be criticised, since it nonetheless equates such a shift with openness and progress. It seems Himanen is unaware of Kuhn's monastic idea of knowledge, which Fuller (2003) clearly points out, despite independently criticising such a monastic attitude in his *Hacker Ethic*:

> The opposite of this hacker and academic model can be called the closed model, which does not just close off information but is also authoritarian. In a business enterprise built on the monastery model, authority sets the goal and chooses a closed group of people to implement it. After the group has completed its

[6]This law is named after the creator of the Linux operating system, Linus Torvalds.

> own testing, others have to accept the results as it is. Other uses of it are "unauthorised uses". We can again use our analogy of the monastery as an apt metaphor for this style, which is well summed up by Saint Basil the Great's monastic rule from the fourth century: "No one is to concern himself with the superior's method of administration". The closed model does not allow for initiative or criticism that would enable an activity to become more creative and self-reflective.[7]

In addition to these similarities between the open-source model and the scientific model, Himanen points out that the scientific model has the same two fundamental obligations as the open-source model, namely, "the sources must always be mentioned (plagiarism is ethically abhorrent), and the new Solution must not be kept secret but must be published again for the benefit of the scientific community. The fulfillment of these two obligations is not required by law but by the scientific community's internal, powerful moral sanctions"[8].

From the Open Society to Free Open Source Software

This section will investigate the similarities between the underlying values of an *open society*, and the *open model* of OSSD. In addition, this section will investigate whether OSS licenses[9] ensure the ethical use, and openness of OSS. The rules of these licenses can be compared to the rules of a constitution in a democratic society, which also protect freedoms, albeit the freedoms of citizens. In addition, just as the state, according to Popper, exists to protect the freedoms of the individual, software licenses are there to protect the freedoms of open source software.

In order to expose the underlying value system of the open source model, it may be informative to consider in more detail the differences between the open and closed models[10] of software development. Himanen (2001) points out that the opposition between the *hacker ethic* and the *Protestant ethic*, is at the heart of the opposition between OSS and proprietary software. He argues: "In stark contrast to this revitalized Protestant money ethic, the original computer-hacker ethic emphasized openness". He also asserts that the proprietary

[7] Himanen (2001) (pp. 70)
[8] (Himanen, 2001) (p. 69)
[9] The all-encompassing term "copyleft" licenses was coined in opposition to "copyright" licenses to represent licenses for OSS.
[10] The reader is reminded that Popper also contrasts open and closed models of social development, as was described earlier in sections 6.1.3 and 6.1.2 respectively.

model promotes *profit over passion*. Another author, Raymond, who is the co-founder of the OSI and appointed representative—to the press, business and public—of the open source community, also provides a distinction between these two contrasting modes of software development: he terms the closed model, the *cathedral* model, and the open model, the *bazaar* model (Raymond, 1999):

> Raymond defines the difference between Linux's open model and the closed model preferred by most companies by comparing them to the bazaar and the cathedral. Although a technologist himself, Raymond emphasizes that Linux's real innovation was not technical but social: it was the new, completely open social manner in which it was developed... Raymond defines the cathedral as a model in which one person or a very small group of people plans everything in advance and then realizes the plan under its own power. Development occurs behind closed doors, and everybody else will see only the 'finished' results. In the bazaar model, on the other hand, ideation is open to everyone, and ideas are handed out to be tested by others from the very beginning. The multiplicity of viewpoints is important: when ideas are disseminated widely in an early stage, they can still benefit from external additions and criticisms by others whereas when a cathedral is presented in its finished form, its foundations can no longer be changed.[11]

Raymond's distinction between the cathedral and bazaar models, can be paralleled with Popper's distinction between closed and open societies. Indeed, many values of an open society are evident in Raymond's quotation. First, is the *pluralistic* nature of the open source model in which "a multiplicity of viewpoints is important", and "ideation is open to everyone". Underlying this approach is the ethic of respect for individual contributions. Pluralism, the free exchange of ideas, and decentralised inspection naturally lead to *critical public debate* and *creative problem solving*, which, in turn, ensure the software product is more accurately assessed as to its correct functioning. Second, is their common *rejection of a blueprint method*. This means that both are anti-authoritarian, unlike the cathedral model Raymond describes. Related to authority, just as Popper argues that it should be possible to replace leaders peacefully and without bloodshed, the open source community believes

[11] Himanen (2001) (pp. 66-67)

that "the hacker network's referee group retains its position only as long as its choices correspond to the considered choices of the entire peer community. If the referee group is unable to do this, the community bypasses it and creates new channels"[12]. Furthermore, it is well known that Linus' group (the creators of Linux) "does not, however, hold any permanent position of authority"[13]. Finally, Himanen states that "Hackers have always respected the individual. They have always been anti-authoritarian".

Despite the self-organising and decentralised nature of OSS teams, many believe that a central manager—even if the person is a volunteer—is essential for OSSD. In a similar way, Popper believed that a state is necessary to protect the freedoms of the individual.

It is conjectured that an important way of ensuring both the ethical use of OSS, and the openness of the OSS model, is through *licensing mechanisms*. As discussed in section 3.3.3, Dafermos (2004) not only believes there is a causal link between Popper's philosophy and FOSS but also emphasises that OSS licensing, paired with Popper's concept of openness, promotes an ethical attitude towards technology and ensures a more socially accountable community. He argues that "The vantage point that is so critical to the FSF, and which it seeks to foster, is that of social responsibility in an open society". He believes that licensing mechanisms such as the GNU General Public License (GPL)—arguably the most 'open' among all FOSS licenses—protect the openness of OSS software, and promote its ethical practice and use. The GNU GPL, designed originally by Stallman as an ethical device, is endorsed by both the FSF and the OSI. The FSF in particular, deem it a license which fosters socially responsible behaviour toward technology. The author specifically considers two alternative OSS licenses (mainly because the advances in technology since the GNU GPL was created in 1984, have meant that the license is somewhat outdated by today's standards), namely, the Greater Good Public License (GGPL), which has been recently re-named the Common Good Public License (CGPL), and the Hacktivismo Enhanced Source Software License Agreement (HESSLA). CGPL caters for digital freedom and human rights, and is concerned with preserving our natural co-habitat, by limiting the environmentally-unfriendly uses of technology. HESSLA exists to develop and deploy computer software

[12]Himanen (2001) (p. 72)

[13]Himanen (2001) (p. 65). Despite the self-organising and decentralised nature of OSS teams, successful large OSS projects almost always have a central chief architect, or a central committee of architects, whose role it is to protect the integrity of the OSS product. In a similar way, Popper believes that a state is necessary to protect the freedoms of the individual. The OSS architects are still accountable to their community (in much the same way as the state is accountable to its citizens), but they wield considerable power, often kept in place by the technical respect that the community has for them.

that promotes the fundamental human rights and freedom worldwide of end-users. It uses the United Nation's definition of human rights as a basis and forbids the use of software for purposes that infringe upon human rights. Its primary function is political—to promote human rights—and its secondary function is legal—to ensure that licenses are legally enforceable. Finally, in order to facilitate the achieving of a common understanding of ethics, the author develops a taxonomy of ethical licensing. He concludes that OSS licensing mechanisms are essential since "Even if one is certain of what constitutes ethical and unethical, one still can't police the ways technology will co-evolve with, and be shaped by individual end-users" (Dafermos, 2004).

Not only does the OSSD approach share values in common with an open society, but the Internet itself—the entire toolbox of e-mail, mailing lists, newsgroups, webpages, file servers and so on—can be seen as enabling an open society. As Himanen argues:

> The Internet does not have any central directorate that guides its development; rather, its technology is still developed by an open community of hackers. . . Interested people on the Internet provided the feedback, stimulation, ideas, source-code contributions, and moral support that would have been hard to find locally. The people on the Internet built the Web, in true grassroots fashion.

The Internet is essential to OSSD, and enables the creation of complex systems such as Linux. The teams who collaborate via the Internet form a loosely-knit group of distributed volunteers. There is usually not an emphasis on central planning[14], nor on financial incentives[15].

Finally, Moroz (2008) lists some key elements, which are common to both OSSD and the open society. These elements, which were briefly mentioned in the "Related work" chapter, are:

- Fallibility of knowledge
- Responsiveness
- Transparency
- Pluralism and multi-culturalism

[14]Some aspect of coordination and planning, even if it is just a list of the desired features that volunteers are requested to work on, and an intended next release date and its contents, usually exists.
[15]The incentives are usually related to seeking peer recognition, and a reputation in the field.

- Responsibility
- Freedom and human rights
- Social mobility and a matter of openness

The investigation in this appendix has shown that there are many similarities between OSSD and the Popperian ideas of *evolutionary epistemology* and *open society*. While these findings do not have direct relevance to the topic of this dissertation, which focuses specifically on Agile software development, they do point to a potential topic for future research.

Appendix G

The Open Source Definition

The Open Source Initiative (OSI) created the following Open Source Definition[1] to determine whether a software license[2] can be considered open source or not:

1. **Free Redistribution**

   The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.

2. **Source Code**

   The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.

3. **Derived Works**

   The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

4. **Integrity of the Author's Source Code**

[1] Coar (2006)

[2] The GNU General Public License (GPL) is one of the most popular examples of an open source software license.

The license may restrict source-code from being distributed in modified form *only* if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.

5. **No Discrimination Against Persons or Groups**

   The license must not discriminate against any person or group of persons.

6. **No Discrimination Against Fields of Endeavor**

   The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

7. **Distribution of License**

   The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

8. **License Must Not Be Specific to a Product**

   The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.

9. **License Must Not Restrict Other Software**

   The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.

10. **License Must Be Technology-Neutral**

    No provision of the license may be predicated on any individual technology or style of interface.

Appendix H

The Free Software Definition

The Free Software Foundation (FSF) defines free software in FSD (2008) by whether or not the recipient has the freedoms to:

- run the program, for any purpose (freedom 0);
- study how the program works, and adapt it to your needs (freedom 1);
- redistribute copies so you can help your neighbor (freedom 2); and
- improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3).

According to the FSF: "Access to the source code is a precondition" for freedoms 1 and 3.