THE INDUSTRIALISATION OF SOFTWARE PRODUCTION
– A KNOWLEDGE MANAGEMENT PERSPECTIVE

MELCHIOR JACQUES VAN NIEKERK

Thesis presented in fulfilment of the requirements for the degree of
Master of Philosophy
(Information and Knowledge Management)

STELLENBOSCH UNIVERSITY

Supervisor: DF Botha

March 2009

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly or otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: 23 February 2009

Acknowledgements

Lindy, if not for your love and support, I would never ever have finished this. Thank you.

En vir my ouers – sonder hulle vertroue en ondersteuning sou ek nooit eers hiermee begin het nie.

And finally I would like to express my gratitude to my supervisor, Daan Botha - for his patience, encouragement and willingness to share knowledge.

Abstract

This research utilises theories of organisational knowledge creation from the field of knowledge management to analyse the manner in which the industrialisation of the software development industry is likely to occur. The aim of the research is to prove the following hypothesis:

If the software development industry moves towards industrialisation, then knowledge assets in the format of universal production templates will come into being.

The research commences by providing background information on the state of practise of software engineering by giving an overview of the changes in the industry over the past four decades.

The software development industry is consequently presented from the viewpoint of the proponents of a craftsmanship based approach to software development, and from the viewpoint of those proposing that industrialisation will offer a solution to the problems besetting the industry. In this discussion the terms industrialisation as well as economies of scale and scope are defined. Potential paths and drivers that will allow the industrialisation of the industry are presented – software factories as a path towards industrialisation, and cloud computing as a driver for industrialisation.

In order to supply a knowledge management perspective, the theories of Ikujiro Nonaka and Max Boisot are presented. These theories assume different perspectives on the creation of organisational knowledge, but an attempt is made to reconcile the differences between the two theories. Particular attention is paid to the economic meaning and implications of knowledge, information and data as factors of production. The concept of knowledge assets are examined in detail, and placed into the context of software development.

In the last chapter the research and conclusions of the previous chapters are consolidated, to prove the central hypothesis of this work.

Opsomming

Hierdie navorsing gebruik teorieë oor kennis skepping in die organisasie vanuit die studieveld van kennis bestuur, om 'n analise te doen van die wyse waarop industrialisasie in die sagteware ontwikkelings industrie moontlik kan plaasvind. Die doel van die navorsing is om die volgende hipotese te bewys:

Indien die sagteware ontwikkelings industrie na industrialisasie toe beweeg, dan sal kennis bates in die formaat van universele produksie patrone onstaan

Die navorsing begin deur agtergronds inligting aangaande die toestand van die sagteware ingenieurswese praktyk te gee, by wyse van 'n oorsig van veranderinge in die industrie oor die afgelope vier dekades.

Die sagteware ontwikkelings industrie word daaropvolgens weerspieël vanuit die oogpunt van voorstanders van 'n handbedrewe benadering tot sagteware ontwikkeling, en vanuit die oogpunt van diegene wat glo dat industrialisasie 'n oplossing kan bied vir die probleme in die industrie. In die loop van hierdie bespreking word die terme industrialisasie, sowel as die van skaal- en bestek ekonomie gedefiniëer. Potensiële roetes na industrialisasie en drywers vir industrialisasie word voorgelê – sagteware fabrieke as 'n roete na industrialisasie, en *cloud computing* as 'n drywer vir industrialisasie.

Ten einde 'n kennis bestuurs perspektief te verskaf, word die teorieë van Ikujiro Nonaka en Max Boisot voorgelê. Hierdie teorieë veronderstel verskillende perspektiewe op kennis-skepping in die organisasie, maar 'n poging word aangewend om die verskille in die twee teorieë met mekaar te versoen. Besondere aandag word gegee aan die betekenis en ekonomiese implikasies van kennis, informasie en data as produksie faktore. Die konsep van kennis bates (*knowledge assets*) word in detail ondersoek, en word bekyk in die konteks van sagteware ontwikkeling.

In die laaste hoofstuk word die navorsing en gevolgtrekkings uit die vorige hoofstukke gebruik om die kern-hipotese van hierdie werk te bewys.

Table of Contents

List of Figures

Chapter 1
Introduction

1.1 Background

Software applications range in scale from enterprise level resource management systems to desktop utilities that allow one to listen to music. The application domains in which software is used range from the space industry to the entertainment industry to household budget management. The tools used to develop software applications range from machine level coding instructions to code-generating frameworks. Software is deployed on platforms that range from cellular phones to supercomputers. It is used by everyone from pre-school children to Nobel Laureates. Software is ubiquitous. One would expect that the production of such a basic commodity would rely on well established principles and processes, and indeed one would expect the software development industry to exhibit some of the hallmarks of industrialisation, notably economies of scale.

In practice, the software industry has long been subject to the criticism that software development projects very often do not meet budgetary and deadline constraints. Software projects often fail to meet the functional requirements set for the project, even if the project is completed within the planned timeframe. These failures have been well documented. A representative sample of failed projects was given in an article published by in IEEE Spectrum Online[1]. These failures include:

Year	Company	Outcome
2005	Hudson Bay Co.	Inventory system problems contribute to $33.3 million loss
2004	Avis Europe PLC	ERP system cancelled after spending $54.5 million
2004	J Sainsbury PLC	Abandoned supply chain management system after spending $527 million on deployment costs
2002	McDonald's Corporation	Information-purchasing system cancelled after having spent $170 million.

[1] Charette, R.N. 2005. *IEEE Spectrum Online.*

A variety of reasons are offered by the industry as justification for such overruns and deficiencies – poorly defined scope of projects, changing scope of projects, resource constraints and technological constraints, amongst others. Given the very large number of software projects that are planned and executed every year, there are always exceptions to the rule, but the exceptions seem to reinforce the view that software production does not enjoy the advantages of industrialisation.

If the industry enjoyed the advantages of industrialisation, these advantages would have been reflected in lower production costs, higher quality, greater adherence to project time lines, and the ability to utilise economies of scope.

The apparent lack of industrialisation in the software industry, and the consequences for the industry if such industrialisation was to take place, is the subject of this study. The study is conducted from a knowledge management viewpoint, and draws on theories and ideas in the field of organisational knowledge creation to prove the hypothesis that is proposed in this study.

1.2 Research Domain

Since the application domain of software development is so large, it is necessary to concentrate on specific classes of software development domains.

It is possible to classify software according to the scale of the projects and the application domain in which the software will be used.

The scale of software development projects range from hobbyist projects, to systems that are used to manage transcontinental telecommunications systems. Hobbyist projects are typically confined to a single person, and developed to suit the needs of the individual. Such projects do not require formal project plans and budgets. On the other end of the scale are projects on which thousands of developers are employed, and with correspondingly large budgets. Guah provides a breakdown of the attributes of very large scale information technology projects[2]. According to Guah, a project can be defined as a very large scale project when the project costs exceeds five million United States dollars, when the duration exceeds 36 months, when more than 500 end users are affected by the project, and when more than ten business units are affected. He lists a number of other attributes, and admits that these attributes give only an approximate

[2] Guah, M. 2008. *Managing Very Large IT Projects in Businesses and Organizations* p. 3-4

means of gauging the size of a project. Using the same set of attributes, it is possible to define small, medium and large projects. The scale of projects that are referred to in this work will be positioned anywhere from medium to very large scale projects.

In terms of the application domain, the study implicitly refers to enterprise-type projects. These are projects that are undertaken to support business initiatives, operations and strategy within commercial organisations. The actual domain within which the enterprise operates is not relevant, but specific and highly specialised scientific applications are, for example, not considered here. Industrialisation is not limited to enterprise type of projects, and all software development would potentially be affected by industrialisation, but all references in this study assume that the domain of interest is enterprise projects.

Enterprise-type development refers to software development projects that are intended for use within an organisation, to support specific functions in the organisation. This class of software application is often required to integrate with existing software applications within the organisation, and these applications are typically used by a relatively large number of concurrent users. Enterprise applications are frequently developed in-house, but are sometimes outsourced to professional software development consultancies. This type of application is always based on the specific requirements of the organisation, and the progress of the project is always measured against a project plan that is owned by the organisation. The applications normally have clearly defined functional requirements, as well as requirements for performance and reliability.

Enterprise applications form part of the infrastructure of the organisation, and such applications have become as indispensable to the day-to-day operation of the majority of large organisations, much as plumbing and electricity are[3] indispensable.

The tools and methodologies that are used to create enterprise applications are well documented and understood. Enterprise applications typically encapsulate business-level functionality, and are therefore to a large degree decoupled from the hardware on which the applications are deployed. The latter is an important consideration for the analysis of the software development process – it is known[4] that the production of electronic components (specifically computer processors or micro chips) is a highly-

[3] Clemons, E.K., Row, M.C.1991. *MIS Quarterly* p. 289-290
[4] Greenfield, J. Short, K. 2004. *Software Factories* p. 5-6

industrialized process that takes full advantage of economies of scale. All computer electronics involve software to some extent, and it is important to maintain the distinction between the software that is embodied in the electronics, and that which exists at a much higher level of abstraction. By concentrating on enterprise applications the distinction becomes quite clear.

Returning to the process of industrialisation, it appears that the software industry has not reached the level of industrialization exhibited by other *infrastructure* industries. There is a consensus in the software industry that the current approach to application development displays more similarities to a craft than to an industrialised process. Although there are software specialists who ply their trade only in specific business domains (consider for example ERP systems programmers who develop only human resource management related software), the majority of development skills reside with programmers who work in various, widely differing business domains. They use their skills to implement the systems that have been specified by specialist analysts within the business.

The central hypothesis that this study seeks to prove or disprove is presented in the next section.

1.3 Hypothesis

The aim of this thesis is to examine the process of software industrialisation, paying specific attention to knowledge management aspects of the process. In the process the work will attempt to shed light on the issues surrounding the industrialisation of this important industry, and will present an approach that allows the knowledge embodied by the industry to be analysed as a knowledge asset.

This thesis considers the history of software development and the current state of practice in software development. From this overview the idea that the software industry has not been industrialised is developed, and supporting evidence is provided for the possibility that the industry can be industrialised. The supporting evidence includes a proposal that certain paths and drivers exist that could encourage the process of industrialisation. At the same time the study points out differences in the outcome of industrialisation for software production as compared to the industrialisation of other industries. Once the possibility of industrialisation has been established, this is placed into the context of a knowledge management approach to the study of industrialisation.

To this end aspects of organisational knowledge creation are drawn into the discussion, and the assets which will arise from the industrialisation process are identified as knowledge assets, in the sense that the term is used by Max Boisot.

The aforegoing leads to the formulation of the following hypothesis that will be developed in the study as described in the previous paragraph:

> *If the software industry is moving towards industrialisation then knowledge assets in the format of universal production templates will come into being*

The term knowledge asset refers to Max Boisot's definition of the concept: *knowledge assets are those accumulations that yield a stream of useful services over time while economizing on the consumption of physical resources*[5].

It will be shown that the software industry cannot be considered to be industrialised, but that the potential for industrialisation exists. Secondly it will be shown that, in the process of industrialisation, knowledge assets will come into being. It will be possible to analyse these assets using Boisot's model of the I-Space.

1.4 Research Methodology

The research methodology used in this study is based on a study of the available literature that is relevant to the hypothesis that the thesis seeks to support. The hypothesis that the study seeks to prove requires a theoretical analysis of the available literature, because the formulation of the hypotheses does not lend itself to empirical research. This statement is discussed in the next section.

1.5 Empirical versus Theoretical Research

The hypothesis as formulated above comprises statements based on abstract theoretical knowledge, and assumptions about the future.

In the first instance, the idea that the software industry can be industrialised depends on the definition of industrialisation, as well as on the current state of the software industry. In order to prove the hypothesis, it is not necessary to prove that the software industry is industrialised, but simply to prove that it can possibly be industrialised. The state of the industry at the current moment is not a matter for empirical research – the state of the industry here is relevant only insofar it is relevant to the possible industrialisation of the

[5] Boisot, M., H. 1998. *Knowledge Assets* p. 13

industry. Additionally – the software industry is a global industry, and the highly diffusible nature of the tools used in software development ensures that the technology used in different countries is very similar[6]. Since a significant body of academic and peer-reviewed literature exists that addresses both the problems in the industry as well as the challenges facing the industry, it is possible to synthesize a view on the state of the industry by referring to these studies.

Secondly – since the software industry has not been industrialised (as is shown in chapter four), it is not possible to conduct an empirical study of how the industrialisation manifests itself. Instead the study relies on proposals for technology that could feasibly promote industrialisation in the industry in order to support the part of the hypothesis that states that universal production templates will come into being as a consequence of the process of industrialisation.

Finally – the hypothesis states that the production templates mentioned above will be knowledge assets. Knowledge assets are themselves abstract concepts that result from the interpretation of Max Boisot's theory of organisational knowledge creation as derived from the I-Space framework. The I-Space framework and the theory of organisational knowledge creation have not been empirically proven[7], and are therefore not tractable to empirical analysis.

The reasons given above shows that an empirical study cannot be undertaken to prove the stated hypothesis. A notional research methodology has therefore been applied to this study, using concepts and abstract ideas to support the hypothesis.

1.6 Literature Study

The literature study contains elements of three distinct areas of study - these areas of study consist of techniques and processes of software development, the economics of industrialisation and theories of organisational knowledge creation.

In the first instance the study examines the creation of software itself. It looks at the existing methodologies, techniques and tools that are used during the production of

[6] This issue is discussed in further detail in chapters five and six. Here it is sufficient to notice that software tools can be distributed electronically (and that there are virtually no barriers to such distribution). If a specific country does not have access to electronic distribution channels, it is axiomatic that such a country cannot be developing software that has any significant role to play in the industry, since the modern software industry is entirely reliant on electronic distribution channels.

[7] Boisot, M. H. 2007. *Explorations in Information Space* p.218

enterprise software applications. The study analyses the reasons why software production is difficult, and then discusses the software development life cycle in terms of process models and design paradigms that are encountered in enterprise application development.

The second area of study is that of industrialisation and the relationship between industrialisation and the software industry. The literature study defines the concept of industrialisation, and its manifestations in achieving economies of scope and scale. The work proceeds by drawing a conclusion on the state of practise in software engineering, and relating this to whether software production may be regarded as a craftsmanship based industry, or an industrialised industry. The possible manifestations and drivers of industrialisation within the industry are analysed by presenting literature that describes instances of these manifestations and drivers.

The third dimension of the study encompasses theories of organisational knowledge creation. The main sources for this part of the study are the works of Max Boisot and Ikujiro Nonaka. The theories of these academics are presented in depth, and related to each other to provide an integrated view of the two theories. From a knowledge management perspective, the analysis proceeds by examining the process whereby tacit knowledge is codified and abstracted. It shows that the process of codification and abstraction is necessary to the industrialisation of the software development industry, and furthermore, that this process will give rise to knowledge assets, in the idiom of Boisot as described in his book *Knowledge Assets*[8]. The knowledge management analysis utilises Boisot's concept of the I-Space to show how the development of an industrial approach to software development will lead to a specific social learning cycle being made manifest in the I-Space, at the level of the industry itself.

This thesis will attempt to shed some light on the issues surrounding the industrialisation of the software development industry, and will present an approach that allows the knowledge embodied by the industry to be analysed as a knowledge asset.

The research is presented in three parts, corresponding to the fields of study enumerated above. Chapter three looks at the history and current state of software development techniques. Chapter four correlates ideas about industrialisation with the state of the software development industry. Chapter five takes an in-depth look at theories of

[8] Boisot, M., H. 1998. *Knowledge Assets*

organisational knowledge creation. The final chapter, chapter six, integrates the results of the literature study, and presents an argument in support of the hypothesis this work seeks to support.

1.7 Conclusion

This chapter has provided an overview of the field of study of this research, and has described the research methodology. A justification has been given for using a theoretical or notional approach to the research, rather than an empirical approach.

The next chapter gives an overview of the pertinent literature that has been used to support the study.

Chapter 2
Literature Overview

This chapter provides an overview of the literature on which this research draws. The overview categorises the literature according to the specific aspect of the research that has drawn on the particular works. This overview does not provide an analysis of the literature, but serves to motivate the sources used, and to place these sources into the context of the study.

2.1 Software Development

The chapter on software development provides an overview of the history of software development and gives a historical perspective on the challenges faced by the industry. The chapter also presents aspects of the software development life cycle, with particular reference to current process models and design paradigms. The aim of the chapter is to sketch the current state of practise in the software development industry.

In the first part of the chapter the challenges faced by the software industry are introduced by referencing a paper published by F.P. Brooks, in a 1987 edition of the IEEE Journal *Computer*[9]. This paper was widely referenced as a summary of the problems facing the software industry at the time of its publication. The paper provides a basis that can be used to formulate the challenges that the industry is facing. Based on Brooks' publication the four challenges that are faced by the discipline of software engineering are identified and analysed in chapter three. The four challenges are those of complexity, conformity, changeability and invisibility. These challenges are discussed with reference to the *Software Engineering Body of Knowledge*, which is a guide published by the IEEE Computer society with the aim of establishing *a baseline for the body of knowledge for the field of software engineering*[10].

No discussion on the history of modern software development can be complete without reference to the unified modelling language. The unified modelling language plays an important role in software development and the development of the UML itself has impacted the software development industry. The *Unified Modeling Language User*

[9] Brooks, F., P. 1987. *Computer*
[10] Abran, A., Moore, J., W. (eds). 2004. *Guide to the Software Engineering Body of Knowledge* p. vii

Guide has been used as a reference for placing the unified modelling language into the context of the industry. The guide was authored by Booch, Rumbaugh and Jacobson, each of whom was instrumental in the evolution and design of the unified modelling language. Greenfield's *Software Factories*[11] is used to provide additional background on the history of software development in this section.

The rest of the chapter is devoted to a discussion of aspects of the software development life cycle. The discussion of the software development life cycle includes architectural and operational aspects of software development. Albin's *The Art of Software Architecture*[12], which provides an overview of and guidelines for the practise of software architecture, is used as a source for describing the different views on software architecture that various stakeholders adopt. The author is associated with both the ACM and IEEE Computer and Engineering Management societies.

Messerschmidt and Szyperski's *Software Ecosystem* provides a context for relating software to the real world – it relates the technical and non-technical issues involved in the software development process, and places these issues into the context of businesses and practical applications. The book has been used as a reference to move aspects of the software development process into the context of the larger world that surrounds the pure software development industry.

The software development life cycle is discussed under two sub-topics. The first of these is process models, and the second is design paradigms. Process models refer to the processes which are used to drive the software development life cycle, and design paradigms refer to the software design principles that are employed when designing software.

The process models specific to the software development industry are categorised as either *waterfall* or *iterative* processes. The waterfall process was first described by Royce, in the Proceedings of IEEE Wescon in 1970[13]. The waterfall process has fallen out of favour in the development community, due to problems inherent in this process. A list of these problems and commentary on the current state of use of the waterfall process were obtained by referencing Marasco's *The software development edge*[14]. Joe

[11] Greenfield, J. Short, K. 2004. *Software Factories*
[12] Albin, S., T. 2003. *The art of software architecture*
[13] Royce, W. 1970. *Proceeding of IEEE Wescon 26*
[14] Marasco, J. 1999. *The software development edge*

Marasco is an experienced project manager and executive with IBM, who has had wide experience in implementing large scale software products.

The iterative approach to software development was developed as an alternative to the waterfall process, because the latter exhibited a number of serious shortcomings, as mentioned above. Variations of the iterative approach are widely used in software development today. The iterative process is discussed with reference to Greenfield and Albin's works, which have been mentioned above, as well as Kroll and MacIsaac's *Agility and discipline made easy.*

Agile development, as the latest widely adopted type of process at the time of writing, is described by referring to the original on-line publication of the Agile Manifesto[15], which lists and describes the motivation and goal of the agile development process. As an example of the implementation of an Agile process, the *Scrum* technique is discussed in depth. The discussion on scrum starts by examining a seminal work by Nonaka and Takeuchi, in which they first described the scrum technique. This work first appeared in an article published in Harvard Business Review[16] (at that time they referred to the technique as the *rugby technique*). The concept of scrum was subsequently refined and further developed by Jeff Sutherland, who explored the concept in a number of papers he published after the publication of Nonaka and Takeuchi's Harvard Business Review article. The discussion on scrum uses Sutherland's publications as the basic reference for describing the process.

Design paradigms in the software development industry currently focuses largely on object oriented design. Object oriented design is explored by referring to Greenfield's *Software Factories*, Booch's reference for the unified modelling language, as well as the widely used reference work *Design Patterns*[17] by Gamma et al. This latter text describes commonly recurring designs in object-oriented software implementations, and is used as a standard reference for software developers.

Greenfield's Software Factories are used to explore future directions in the approach to software design. Although it is expected that object-orientation will always be part of software design, new innovations such as software factories will enable developers to solve design and implementation problems that cannot be solved by object orientation

[15] http://www.agilemanifesto.org
[16] Nonaka, I. Takeuchi, H. 1986. *Harvard Business Review*
[17] Gamma, R., et al. 1995. *Design Patterns*

on its own. This concludes the overview of the literature references for chapter three.

2.2 Industrialisation

Chapter four, *Industrialisation and the Economics of Software Production,* presents industrialisation as a concept, and places it into the context of the software development industry. The chapter utilises definitions from the *Encyclopedia of Business and Finance,* and material from Drucker's *Post-Capitalist Society*[18] to define the concept of industrialisation and to list the ways in which industrialisation has been observed to typically manifest itself in an industry. A closer examination of the industrialisation process in a specific industry, namely that of the microcomputer industry, is done in this chapter. The examination draws primarily on Langlois' 1992 article[19] titled *External Economies and Economic Progress: The Case of the Microcomputer Industry.*

Industrialisation generally leads to economies of scale becoming manifest in an industry. This is not the only possible means by which industrialisation can help an industry to economise on the factors of production – economies of scope is an alternative to economies of scale in this respect. The next section in this chapter defines economies of scale and scope, drawing on the *Encyclopedia of Management*[20] for definitions, and the work of Greenfield for an industry-specific definition of the economies of scope.

The state of practice of the software development industry is analysed next. This chapter argues that there existed, and still exists, a series of problems within the software industry. In order to support this viewpoint, a number of different publications are referenced. By comparing studies from the late 1960's, represented by the NATO conferences of 1968 and 1969, reports from the mid 1990's, and finally reports from 2003, it is possible to infer the state of practice in the industry and obtain an impression of the progress that has been made over a period of four decades. The reports on the two NATO-sponsored conferences on software engineering that was held in Brussels in 1968 and 1969 form the starting point for the discussion. According to McClure the *motivation for these conferences was that the computer industry at large was having a great deal of trouble in producing large and complex software systems*[21]. The original

[18] Drucker, P., F. 1993. *Post-capitalist society*
[19] Langlois, R., N. 1992. *The Business History Review*
[20] Koch, J., C., Inman, A., R. 2006. *Encyclopedia of Management 5th ed*
[21] McClure, R.M. 2001. *Online*

papers were published after the NATO sponsored conferences, and are currently available on-line[22] [23].

The primary reference for the state of practice in the 1990's is an article published by W.W. Gibbs in a 1994 edition of Scientific American[24]. In the aforementioned article Gibbs described a *chronic crisis* in software development, based on research and interviews with a number of sources. The state of practice in 2003, is derived from a series of articles published in the 2003 edition of IEEE Software, volume 6[25]. The series of articles were written in order to produce an overview of the state of practice in software engineering. Each of the articles focus on a different aspect of the state of practice. The series of articles considers the amount of progress that has been made towards solving the problems that had previously been identified in the industry.

Software development may be considered to be either a craft based industry which is somehow not susceptible to industrialisation, or it may be considered to be an industry that can feasibly be industrialised.

The view of software development as craft is explored with reference to McBreen's 2001 publication: *Software Craftsmanship: The New Imperative*[26]. McBreen's conclusions are supported by a number of sources, including Cockburn and Highsmith[27] writing in Computer journal, and the documented experience of Coplien at Borland[28]. The viewpoints of these authors support the notion that software development should be crafts-based, and essentially reject the idea that industrialisation is a viable option for this industry.

The alternative view, namely the view that the software development industry can potentially be industrialised, is examined from two perspectives – that of the producers of software, and that of the consumption of software in a manner that will encourage industrialisation. The industrialisation of software production is envisioned by Greenfield in *Software Factories.* This work forms the basis of reference for the section on the software factory approach to development. An overview of Greenfield's vision for the industrialised production of software is given in this chapter in order to show

[22] Naur, P., Randell, B. 1969. *Online*
[23] Buxton, J.N., Randell, B. 1970. *Online*
[24] Gibbs, W., W. *1994. Scientific American* p. 86-95
[25] 2003. *IEEE Software*. 20(6)
[26] McBreen, P. 2001. *Software Craftsmanship*
[27] Cockburn, A., Highsmith, J. 2001. *Computer 34*
[28] Coplien, J.O. 1994. *Proceedings of the 5th annual Borland International Conference*

that valid technical approaches can be formulated whereby the industrialisation of the industry can be made feasible.

A driver towards the industrialisation of the industry can be found in the concept of *cloud computing*. The overarching idea of cloud computing is a new concept, although the constituent parts of the idea has been around for a longer period. The discussion on cloud computing is based on a series of articles that has been published by Gartner Research, and which address different operational, architectural and implementation issues for cloud computing.

2.3 Knowledge Management Theory

Knowledge management theory in this research relies on the work of Ikujiro Nonaka and Max Boisot.

Ikujiro Nonaka has published a range of papers (in association with other authors) that describe what was a new perspective on organisational knowledge creation at the time of the initial publication of the paper. Nonaka's approach emphasized Eastern rather than Western values and philosophy. Nonaka started exploring his theory of organisational knowledge creation in a 1991 article published in the Harvard Business Review called *The Knowledge-Creating Company*. This article helped to popularise the notion of *tacit knowledge.* Nonaka expanded on this paper with a series of publications. The first statement of his theory was published as *Theory of Organisational Knowledge Creation* in 1995. This publication was the first to introduce the concept of SECI (an acronym signifying a cycle of socialization, externalisation, combination and internalisation in the knowledge creation process). Further publications introduced the concept of *Ba* in a paper titled *SECI, Ba and Leadership: a Unified Model of Dynamic Knowledge Creation*[29]. The final refinements (at the time of writing) modified the model to consider the organisation as a dialectic being[30], and introduced the concept of group tacit knowledge[31].

The publications by Max Boisot on which the research draws most heavily, include *Information Space: A framework for learning in organizations, institutions and culture,* published in 1995, as well as *Knowledge Assets: securing competitive advantage in the*

[29] Nonaka, I., Toyama, R., Konno, N. 2000. *Long Range Planning 33*
[30] Nonaka, I., Toyama, R. 2003. *Knowledge Management Research and Practice 1*
[31] Erden, Z. von Krogh, G., Nonaka, I. 2008. *Journal of Strategic Information Systems*

information economy, published in 1998. The first mentioned of these two books defines the concept of the I-Space as a framework within which organisation knowledge creation can be studied. Boisot followed this book by a study of knowledge assets within the I-Space: *Knowledge Assets: securing competitive advantage in the information economy*, published by Oxford Press in 1998. These two works provided the pioneering theoretical framework within which to study organisational knowledge creation.

In contrast to Boisot’s work, Nonaka's model is built around empirical observation of the way in which knowledge is created in an organisation, and it does not attempt to provide a theoretical framework. Boisot published a collection of academic papers in 2008, in a volume called *Explorations in Information Space: knowledge, agents and organization.* The collection consists of a set of papers related to Boisot's I-Space framework – each paper is authored or co-authored by Boisot. The theoretical aspects of this research utilises Boisot's I-Space framework as presented in these three books.

2.4 Conclusion

The literature used in this work, addresses the three main areas of study from which the thesis draws. The most important references from the knowledge management domain are the works of Boisot and Nonaka that provide theories of organisational knowledge creation. Chapters three to five of this study provide the underpinnings for the discussion in Chapter six that proves the central hypothesis that the study seeks to support. The final chapter draws on the conclusions and research of the previous chapters to conclude this research.

Chapter 3
Software Development Techniques, Methodology and Tools

This chapter describes the particular management and implementation issues related to software development, and provides insights into the history of software development, and the techniques of software development that are currently being employed.

3.1 The challenge of creating software

In an article written in 1984[32], Frederick Brooks enumerated the challenges facing software engineers at the time. These challenges include:

- **Complexity –** software posses an inherent complexity, that increases in a non-linear relationship to the scale of the project.
- **Conformity –** the author states that software needs to conform to a wide range of requirements, which are variously imposed by the functional requirements of the end product, by hardware requirements, and by any number of additional external factors.
- **Changeability** – software is a more *malleable* product than physical artefacts such as cars or buildings, and thus lends itself to change both during the development process, and afterwards, during the lifetime of the finished application.
- **Invisibility** - *software is invisible and unvisualizable*. The author contends that the difficulties of visualizing software, contributes to the difficulties of creating software.

Brooks wrote this article in response to the common perception (at the time) that software projects appear simple and straightforward, but nevertheless regularly devolve

[32] Brooks, F., P. 1987. *Computer*

into projects that fail to meet time, cost and functional requirements. The author went on to enumerate a number of incremental *breakthroughs* in software development tools and methodologies, and concluded by proposing that the only exponential gains in software development efficiency, will be found by better training of designers. The insight that Brooks had, was that software is possessed of a number of essential, irreducible properties, which make software development difficult.

Since Brooks wrote this article in 1984 dramatic changes have taken place in the domain of information technology, not the least of these being the rise of the internet, and the concomitant (and continuing) series of changes in the way software is written and utilised.

Nevertheless the challenges Brooks identified in 1984 largely still apply to software development today. The issues that Brooks identified as being essential issues of software development are being addressed by a variety of means, and with differing degrees of success.

- **Complexity** - abstraction is used to avoid having to deal with a system's complexity, or to at least alleviate the degree of complexity that has to be addressed. Boisot defines abstraction is a process that gives structure to phenomena[33]. The term abstraction is defined in the detailed discussion on complexity below.
- **Conformity –** software applications are as different from each other as the fields of application of software are. The effects of diversity are alleviated by re-using components of systems, and attempting to create generic applications that can easily be adapted for different scenarios.
- **Changeability** – the discipline of project management addresses and manages change.
- **Invisibility –** tools such as the Unified Modelling Language[34] (commonly known as UML) helps designers to visualize the abstractions inherent in software applications.

The rest of this chapter examines Brooks' four inherent, problematic properties of software development in greater depth, and then examines the methodologies and tools that are used to address these challenges.

[33] Boisot, M., H. 1998. *Knowledge Assets* p. 48
[34] Booch, G., Rumbaugh, J., Jacobson, I. 1999. *The unified modelling language user guide* p. 13-35

3.1.1 Complexity

Complexity in the context of software development refers to the opposite of simplicity, not to complex systems. Complexity may be measured post facto by examining the software that is the result of the development process – in this case complexity is a measure of the resources which must be expanded in developing, maintaining, or using a software product.[35] This measure of complexity is a measure of the accidental complexity inherent in software development, and was originally defined by Brooks[36].

Brooks categorised complexity as being either accidental or essential. Essential complexity is inherent in the problem being solved, while accidental complexity arises from extraneous factors, such as the tools that are used to solve the problem.

Accidental complexity, by contrast, arises from the intricacies of the functional requirements for the software, and the development tools used in creating the software – accidental complexity *is an artefact of the solution*[37.] Because accidental complexity arises from the solution, it can be addressed by changing the solution – which may include measures such as adopting more appropriate or efficient different tools or programming languages.

It is Brook's contention that software has a very high level of essential complexity. The level of essential complexity inherent in software arises from the large number of states that a software system can assume, and from the non-linear manner in which the complexity of a software system increases when the size of the system increases. The non-linear increase in complexity versus size is a result of the components of software systems interacting with each other in a non-linear fashion.

Since essential complexity can by its very nature not be removed entirely[38], the effort to reduce complexity must be focused on reducing the level of accidental complexity. The primary tool the engineer has at his disposal for dealing with complexity in software development is abstraction. Abstraction is the process of selectively removing some information from a description, in order *to focus on the information that remains*[39].

Booch et al define an abstraction as *the essential characteristics of an entity that distinguish it from all other kinds of entities. An abstraction defines a boundary relative*

[35] http://sern.ucalgary.ca/courses/cpsc/451/W98/Complexity.html
[36] Brooks, F., P. 1987. *Computer* p. 2
[37] Greenfield, J. Short, K. 2004. *Software Factories* p. 36
[38] http://stevemcconnell.com/ieeesoftware/eic04.htm
[39] Greenfield, J. Short, K. 2004. *Software Factories* p. 42.

to the perspective of the viewer[40]. This implies that it is possible to derive more than one valid abstraction of a given problem, since the boundary of an abstraction is relative to the perspective of the viewer.

In terms of software development, an abstraction is the implementation of a solution for a given problem. In Booch's terms, the solution forms the new boundary for the problem, and the details of the problem become irrelevant to the design process. Abstractions can use other abstractions to compose solutions to other problems.

Combining these two viewpoints, abstraction is used to define a boundary for a problem and to structure the problem by placing it into a specific abstraction. By reasoning about the abstraction, one can reason about the entire class of problems contained within this boundary – thereby reducing the complexity (and incidentally conserving data processing resources)[41].

Greenfield defines an abstraction as *a partial solution developed in advance to the problems in a given domain, which can be used, along with other abstractions, to partially or completely solve multiple problems in the domain*[42].

In these terms, the application of abstractions lessens the impact of accidental complexity, because the abstraction forms a solution to the problem. Since a solution is at hand, the software engineer does not need to concern herself with overcoming complexity, but simply needs to apply the abstraction to the solution domain.

Even though abstraction offers a means of reducing the essential complexity of the development process, it does not provide a *silver bullet* solution. This is well illustrated by considering the following description from the 2004 version of the *Guide to the Software Engineering Body of Knowledge* (SWEBOK):

> *Some requirements represent emergent properties of software—that is, requirements which cannot be addressed by a single component, but which depend for their satisfaction on how all the software components interoperate. The throughput requirement for a call centre would, for example, depend on how the telephone system, information system, and the operators all interacted under actual operating*

[40] Booch, G., Rumbaugh, J., Jacobson, I. 1999. *The unified modeling language user guide* p. 457
[41] Boisot, M., H. 1998. *Knowledge Assets* p. 49
[42] Greenfield, J. Short, K. 2004. *Software Factories* p. 54

conditions.[43]

Abstraction may serve to make it easier to create the components that will form the solution as a whole, but abstraction will not address the emergent property of software as described above – this can only be addressed in the realisation of the software – i.e. the way in which the software is actually implemented. Domain specific languages, which will be discussed later, are an attempt to cater to some degree for emergent properties in software.

3.1.2 Conformity

The conformity property of software development refers to the heterogeneous environments in which software is required to function, and to the multiple and differing requirements imposed on software products.

The conformity issue arises from a number of different root causes. On the one hand is the requirement imposed by the fact that software is a *late arrival* on the scene, and has to conform to existing legal requirements, user expectations, budgetary constraints, existing data formats, and so forth. Another root cause is the nature of functional and non-functional requirements. Functional requirements may stipulate that a particular piece of software has to be able to interface to two widely differing existing software systems. Non-functional requirements may stipulate that the software should be able to function on a variety of different operating systems, for example Mac OS and Windows.

The complexities introduced by certain environmental factors such as legislative and regulatory requirements constitute an irreducible complication in the software development context, since no amount of engineering or design can cater for requirements which can be arbitrarily imposed by agencies that are outside the development or design context. These complexities are not unique to the software development process, but are also present in a large number of other industries, such as the construction and engineering industries.

The challenge of conformity that is specific to the software development industry is focused on the multi-platform issue, and on software-specific non-functional requirements.

Non-functional requirements for software include the following:[44]

[43] Abran, A., Moore, J., W. (eds). 2004. *Guide to the Software Engineering Body of Knowledge* p. 2-3

- **Usability** – a measure of how well the software support the execution of user task
- **Reliability** – a measure of the frequency and severity of defects encountered during the normal operation of the software.
- **Performance** – a measure of how quickly the system responds to stimuli, and how well (optimally) it uses resources in providing the response.
- **Supportability** – a measure of the cost of supporting (and maintaining) the software after it has been delivered to the customer.

Because non-functional requirements may often be imposed on a set of software applications – for example when an organisation sets a performance standard for all the software it uses – it can be regarded as an externally imposed requirement to conform.

There are a number of approaches, both in terms of software design and software architecture that can be taken to lessen the impact of conformity constraints:

- **Model Driven Development** – this is an approach that relies on creating a model of the software, and using this to inform the development process. Because this approach inherently uses a high level of abstraction, it is relatively easy to apply the model to various platforms.
- **Frameworks** – software frameworks provide a level of abstraction from the underlying hardware on which the software is deployed. This makes software more portable across different types of hardware and operating systems.
- **Standards** – by adhering to industry standards, it is easier to achieve interoperability between different systems. Industry standards are universally agreed upon descriptions of communication protocols, implementation conventions, data exchange formats, and other artefacts and features of software systems.
- **Component re-use** – a component is a software solution that is at a high level of abstraction, and that encapsulates a number of functions. Sharing components between systems ensures that the systems agree on certain abstractions.

The advent of the internet and the appearance of distributed web-based systems added additional layers of complexity to the software development process. Many of the industry standards for web-based application are under development, and are subject to

[44] Greenfield, J. Short, K. 2004. *Software Factories* p. 48

change. Different approaches to implementing man-machine interfaces are commonly seen for web-based applications. Different software vendors often align themselves with a particular technology at the cost of interoperability in order to maximise profits.

The challenge of conformity is most often addressed at the level of software architecture, rather than software development. That this is the role of software architecture can be seen by considering the following statement:

> *A central activity of software architecture design is decomposing the system into subsystems (i.e. components) that work together to satisfy the required functionality. The purpose of this activity is to reduce problem complexity into smaller manageable parts.* [45]

Because architecture focuses on the decomposition into pieces *that work together*[46], the development of a system's architecture is a task that lends itself to considering the complexities of conformance requirements.

3.1.3 Changeability

Brooks states that software is subject to a constant change pressure. There is more than one reason for this pressure coming to bear. In part, it is because the system *embodies its function, and the function is the part that most feels the pressures of change.*[47]. Coupled with this is the perception that software can be changed far more easily than could a physical object such as a building or a vehicle.

According to Brooks all successful software gets changed, for a number of reasons. The first reason is that software may be used beyond the original domain and application for which it was created, thereby adding functional requirements to cope with the extended application domains. A second cause of the changeability of software is that successful software frequently outlives the hardware, operating systems and standards for which it has initially been designed. This has some bearing on the issue of conformity – users will expect software to be updated to take advantage of the latest operating systems and hardware and thereby deliver better performance, while still offering at least the same core functionality.

Stepanek lists a number of reasons why software projects are different from other

[45] AlSharif, M., Bond, W., P., Al-Otaiby, T. 2004. *ACM-SE 42* p. 98
[46] AlSharif, M., Bond, W., P., Al-Otaiby, T. 2004. *ACM-SE 42* p. 98
[47] Brooks, F., P. 1984. *Computer* p. 3

projects, and some of these reasons relate directly to the changeability property of software[48]:

- **Technology changes rapidly.** This is exemplified by the fact that the first enterprise application development framework, Sun's J2EE (http://java.sun.com/javaee/index.jsp) was released in 1998, and Microsoft's competing .Net framework, was released only in 2002. This is an indication of the rate of change in technology platforms – less than a decade ago at the time of writing (2007) no-one had any experience whatsoever with enterprise application frameworks, yet they are currently widely used for enterprise application development.
- **Change is considered easy.** Because software is considered to be malleable, and because of the non-physical nature of software artefacts, change is considered to be easy. This ignores the issue of the quality of change – unless a software architecture supports and enables changes to a system, such changes often have far-reaching long-term effects on the stability and usability of the software.
- **Change is inevitable.** This both because of environmental changes (hardware, etc. as described above), and because of the nature of the software development process, which often leads the end-users of systems to refine their requirements during the process of development. This happens because the end-user (referring to the person, agent or organisation commissioning the software) only understands the potential of the software once it is under development.

The changeability property of software is embedded in the life cycle of the final product, but also exists during the requirements definition phase of a project – as mentioned above, the users of a system often refine requirements as they become aware of the potential of a system. The changeability during the requirements definition phase is exemplified by the fact that change management is specifically mentioned in the Software Engineering Body of Knowledge as being key to the success of the software engineering process [49].

Changeability is managed both by the application of sound project management principles that are tailored to the software development process, and by avoiding

[48] Stepanek, G. 2005. *Software Project Secrets: why software projects fail* p. 7-21
[49] Abran, A., Moore, J., W. (eds). 2004. *Guide to the Software Engineering Body of Knowledge* p.2-9

implementation technologies that are not flexible enough to cater for this essential property. Marasco[50] advocates the use of iterative methodologies, rather than the waterfall approach (as discussed below) to compensate for changeability.

3.1.4 Invisibility

De Gyurky[51] states that having a subjective design for a large software system is not enough; it must be possible to communicate the design to the team involved in the construction of the software. This is the importance of visualization. He notes three contributing factors that lead to a complete visualization:

- Visualizing the product architecture
- Visualizing the organisation
- Visualizing the methodologies and techniques

De Gyurky's contributing factors highlight the fact that visualizing a software development project comprises more than visualizing the software – the software design is part of the visualization of the architecture. The organisation, within which the software will be utilised, and the methodologies and techniques that will be used to create the software must equally be visualized in order to provide a complete picture, and communicate the essence to the entire project team. De Gyurky does note that an exception to this rule exists – namely when a team involved in software development is highly experienced and extremely well integrated. The purpose of visualization is therefore primarily to facilitate exact communication of all aspects of the project to the team.

Brooks concurs with this latter statement in his 1984 paper, emphasizing that communicating ideas within the software development team is a particular challenge, because of the difficulty in visualizing the software. Software is abstract, and the functioning of a system is often only fully expressed by running the software – the design does not describe the system in its totality.

A software system has both a static and a dynamic aspect. The static aspect represents the actual code and components of the system – it is a view that emphasizes the

[50] Marasco, J. 1999. *The software development edge* p. 45
[51] De Gyurky, M., S. 2006. *The cognitive dynamics of computer science* p. 15

structure of the system.[52] The dynamic aspect of the system is a view of the system that emphasizes its behaviour.

The only definitive and absolutely unambiguous description of the static view of a system is the code that constitutes the system. Any effort to create a visual representation of the static view of a system will therefore be a summary view of the structure of the system as represented by the code. Such a view will also not representational of all the known information about the system – the summary view reduces the amount of available data. In a similar fashion the dynamics of a system can only be unambiguously described by looking at the actual run-time behaviour of a system – any model that describes the runtime behaviour will involve some level of abstraction or simplification, and will necessarily not represent all information.

Visual representations of systems are based on models of the systems. A model is defined as *a simplification of reality, created in order to better understand the system being created; a semantically closed abstraction of a system*[53]. Languages such as the Unified Modelling Language are visual languages which are used to describe the model of a software system. This definition of a model (which is formulated from a software engineering perspective) corresponds to Boisot's conceptualisation of the way in which data is transformed into information, and that in turn is transformed into knowledge. The process Boisot describes is a process of cognitive simplification[54]. This process is discussed in depth in chapter five.

The 2004 version of *The Guide to the Software Engineering Body of Knowledge* identifies two categories of visualization tools[55], these being comprehension tools, and re-engineering tools. The latter are tools that facilitate the reconstruction and examination of software, and are not relevant to this discussion. The second category of tools include those tools that assist in the human comprehension of programs, and these are precisely the tools that Brooks referred to.

The visualization property has been addressed to some degree since Brooks mentioned it as constituting one of the essential difficulties inherent in the software creation process. The tools and techniques that are used to visualize software will be discussed in

[52] Booch, G., Rumbaugh, J., Jacobson, I. 1999. *The unified modeling language user guide* p. 466
[53] Booch, G., Rumbaugh, J., Jacobson, I. 1999. *The unified modeling language user guide* p. 463
[54] Boisot, M. H. 2007. *Explorations in Information Space* p. 12
[55] Abran, A., Moore, J., W. (eds). 2004. *Guide to the Software Engineering Body of Knowledge* p. 10-2

greater depth in the rest of this chapter. It is worth noting, however, that visualization techniques do not provide unambiguous views of software systems.

Having considered the challenges inherent in the software development process, we next consider the way in which software is developed by looking at the software development life cycle.

3.2 The Software Development Life Cycle

Creating software is a multifaceted activity. A common perception is that software creation is synonymous with programming.[56]

This statement illustrates the fact that the software development life cycle includes a large range of activities, ranging from determining the requirements for a system, to managing the deployment of the software. A variety of disciplines are involved in the software development life cycle. Albin[57] enumerates a number of different perspectives on the software development life cycle, which is required to completely understand the process:

- **Management view –** the view adopted by (project) managers. This view is oriented towards providing information on the progress towards achieving specific goals.
- **Software engineering view** – this view is closest to the idea that software development consists of *programming*, since it incorporates the coding tasks together with design and requirements analysis. This view informs the software engineer and programmer.
- **Engineering design view –** the view focuses on design, rather than implementation. System analysts use this view.
- **Architectural view –** adopted by the systems architect, this view is focused on the design of the application or system, and focuses on how the design drives development. The architectural view itself takes on different perspectives, including deployment, interaction and logical views.

These views complement each other and the synthesis of the four views provides a more complete insight into the software development life cycle than any single perspective can add. Aspects of the software development life cycle will be discussed in the rest of

[56] Messerschmidt, D., G., Szyperski, C. 2003. *Software Ecosystem* p. 67
[57] Albin, S., T. 2003. *The art of software architecture* p. 17-35

this chapter – process models, design paradigms, the role of software architecture, tools and platforms.

3.2.1 Process Models

Software development process models are structures imposed on the process of developing software. Two main types of development process are generally recognised, these being the *Waterfall* approach, and the *Iterative* approach to software development. There exists a number of refinements and variations of each of the approaches, and combinations of waterfall and iterative processes are sometimes used. In addition to the *Waterfall* and *Iterative* approaches, we also discuss the *Agile* approach in this section – although Agile is in some respects a form of iterative development, it is discussed as a separate process model, since the principles of Agile development is very different from that represented by typical iterative processes such as the Rational Unified Process (RUP).

3.2.1.1 Waterfall

The waterfall process was first described by Royce in a paper presented at the 1970 IEEE /WESCON conference[58]. In this paper Royce described a model for software development that is based on the idea that the software development life cycle can be broken down into distinct phases, that can be performed sequentially[59], and where each phase will build on previous phases.

The waterfall process can be illustrated as in the following diagram:

[58] Royce, W. 1970. *Proceeding of IEEE Wescon 26*
[59] Messerschmidt, D., G., Szyperski, C. 2003. *Software Ecosystem* p. 70

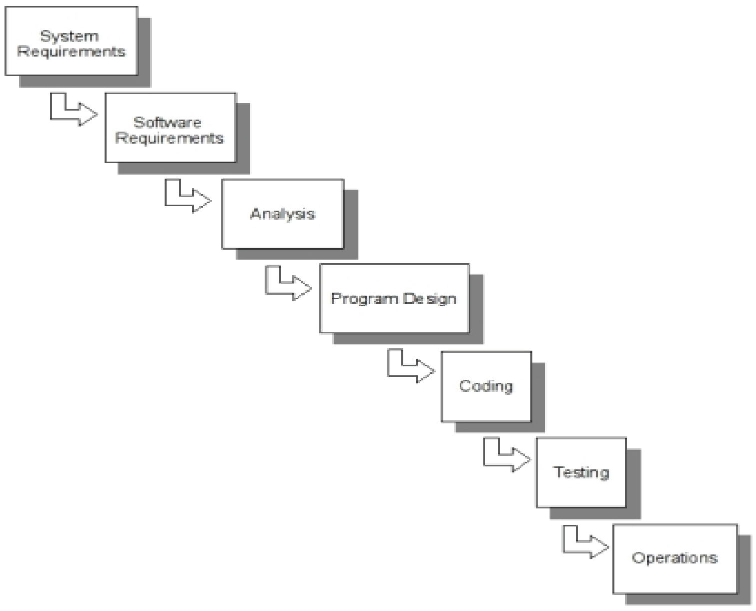


Figure 3.1: The Waterfall Process

Source: Adapted from Royce (1970)

The name of the process model derives from the layout of the diagram, which shows each step as following on to the next step, in a pattern reminiscent of a waterfall. No provision is made in this model for revisiting the work that was done in previous phases – it is assumed that once a particular phase is completed, it will not be necessary to make any adjustments to the output of that phase.

The waterfall approach has fallen out of favour for use in software development[60]. The reasons for this can be traced to the following characteristics of software development projects[61]:

- Resources are scarce – there appears to be little margin for error.
- Requirements are often poorly understood at the beginning of a project.
- Execution errors are unavoidable, especially at the beginning of a project.
- The goal or target of the project almost always changes during the implementation.
- Adhering to a course of action if that course of action proves to be wrong, is *typically catastrophic*.

[60] Marasco, J. 1999. *The software development edge* p. 50
[61] Marasco, J. 1999. *The software development edge* p. 50

The waterfall approach does not address the problems caused by the abovementioned list of characteristics.

The waterfall method *fostered the notion that software development was like a manufacturing process and that the process itself can control the quality of a product.*[62] Software development is very different from a manufacturing process, as has been shown in the first part of this chapter – software development possesses a number of attributes that distinguishes it from manufacturing processes.

An alternative to the waterfall approach is an iterative approach to software development. This approach is described below.

3.2.1.2 Iterative

The iterative process stands in contrast to the waterfall process – rather than developing software sequentially, it breaks the project into iterations. At each stage it captures and implements a subset of the requirements, and refines the work that has been started in previous iterations.[63] Greenfield states that *iterative development distributes requirements definition over many small negotiations*[64]. The process is not of a fixed nature – at any time it is possible to modify requirements, add new requirements, and to feed back the results of a particular iteration into the development cycle.

It is worth noting that Royce in his 1970 article[65] already noted the problems with the waterfall approach, and he advocated an iterative approach in the same paper as a solution to the problems that the waterfall approach represented.

The diagram in Figure 3.2, illustrates the iterative development process. The output of one activity, informs the next activity. Different iterations of the development life cycle allow requirements to be modified.

[62] Albin, S., T. 2003. *The art of software architecture* p. 21
[63] Greenfield, J. Short, K. 2004. *Software Factories* p. 101
[64] Greenfield, J. Short, K. 2004. *Software Factories* p. 98
[65] Royce, W. 1970. *Proceedings of IEEE Wescon 26*

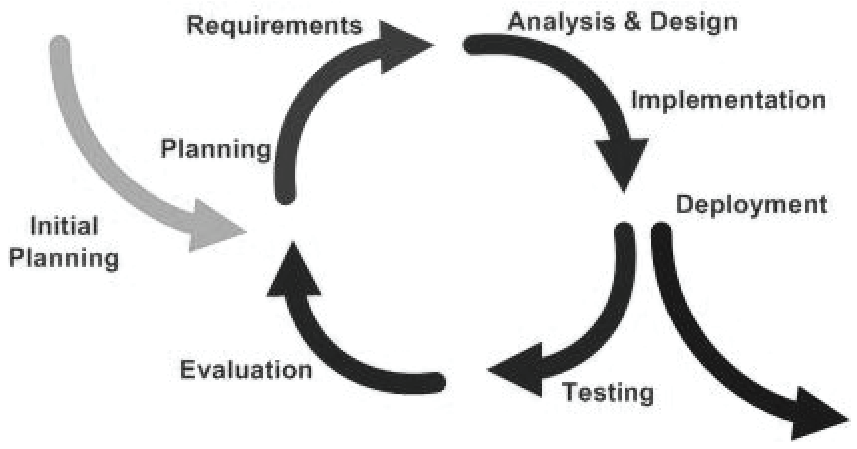


Figure 3.2: The Iterative development model
Source: Wikipedia[66]

There exist a number of refinements of the iterative model – these include the *WinWin Spiral model*[67], and the Rational Unified Process (RUP). The RUP is a particularly well-known member of the family of processes known as the Unified Process family. RUP is tightly coupled to a set of tools that IBM (http://www.ibm.com) sells. Like other Unified Processes RUP is a configurable process, which can be adapted to different scenarios.

The process life cycle as it is implemented in RUP, is shown in the diagram below.

[66] Wikipedia, http://en.wikipedia.org/wiki/Image:Iterative_development_model_V2.jpg
[67] Messerschmidt, D., G., Szyperski, C. 2003. *Software Ecosystem* p. 75

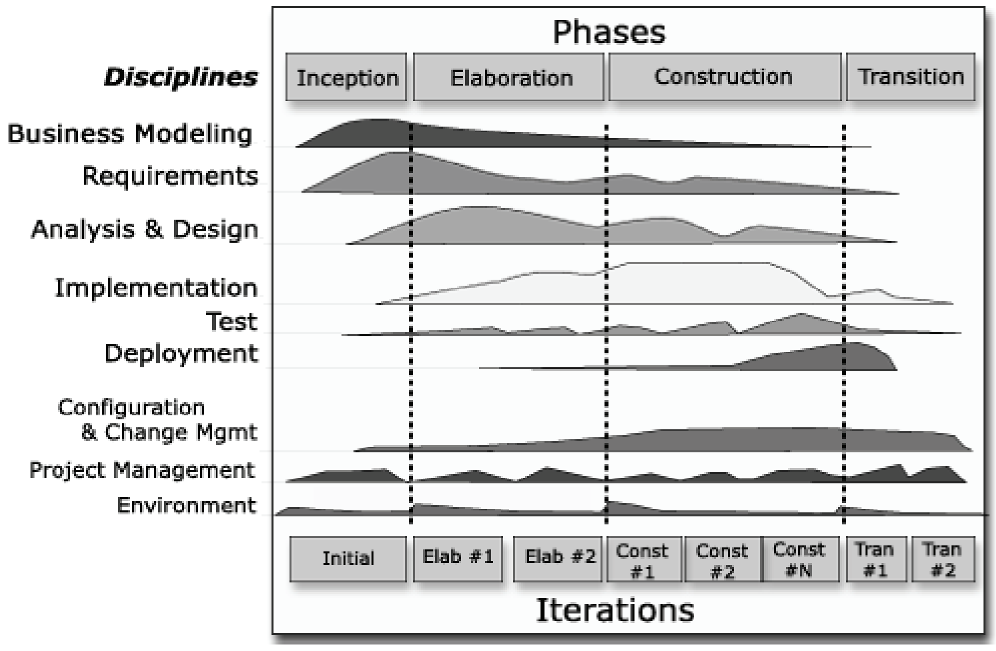


Figure 3.3: The Rational Unified Process

Source: Wikipedia

The Rational Unified Process is an implementation of a Unified Process, but with the difference that it is heavily supported by tools – the process is automated to a large extent.[68] RUP and other Unified Processes are based on six principles which were first defined by the Rational Software Corporation, and later refined by IBM.[69] These principles are:

- Adapt the process.
- Balance stakeholder priorities.
- Collaborate across teams.
- Demonstrate value iteratively.
- Elevate the level of abstraction.
- Focus continuously on quality.

Moreover, all Unified Processes are distinguished by dividing the software development

[68] Kroll, P., MacIsaac, B. 2006. *Agility and discipline made easy* p. 17
[69] Kroll, P., MacIsaac, B. 2006. *Agility and discipline made easy* p. 17

life-cycle into four phases[70]:

Inception – also know as the vision phase, this phase establishes a good understanding of the system, *by reaching a high level of understanding of the requirements*. The output of this phase consists of a product vision, and a business case for the product[71].

- **Elaboration** – also know as the planning and specification stage. The phase involves designing, implementing and testing a baseline architecture. The exit criteria for this phase are a specification of requirements, and an architectural concept[72]. To avoid a waterfall approach, specification is only done to the level necessary to create the exit artefacts, leaving scope for refinement in future iterations.
- **Construction –** the actual construction of the product takes place during this phase. This includes coding and testing. The outputs of previous iterations are adjusted as necessary – specifically the architecture and design. The product may still require fine-tuning in terms of functionality, performance and quality at the end of the initial iteration.[73]
- **Transition –** this phase signals the transition of the product to the users.[74] Testing of the final product and deployment of the product are the main features of this phase. The product may also enter a maintenance and support phase, as a sub-phase of the transition phase.

Apart from the Rational Unified Process, other processes from the Unified Process family include OpenUP which is an open-source process framework.

The general Unified Process, and the process frameworks that are based on it, are representative of iterative process frameworks.

3.2.1.3 Agile

The principles of Agile development are set out in the Agile Manifesto (http://agilemanifesto.org/). The Agile Manifesto states that:

[70] Kroll, P., MacIsaac, B. 2006. *Agility and discipline made easy* p. 12
[71] Albin, S., T. 2003. *The art of software architecture* p. 20
[72] Albin, S., T. 2003. *The art of software architecture* p. 20
[73] Kroll, P., MacIsaac, B. 2006. *Agility and discipline made easy* p. 12
[74] Albin, S., T. 2003. *The art of software architecture* p. 21

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.

Agile processes are frameworks, which enable the creation and customization of processes[75]. The *agility* of the process stems from the fact that processes which have shortcomings are immediately adjusted to fix those shortcomings.

Agile places an emphasis on continuous interaction with all stakeholders, and continuous correction and feedback into the process. The structures and processes within Agile arise from the interactions and team – the teams are self-organizing[76]. Agile is sometimes thought to be unstructured, but this is not true. Although Agile processes do not stand on ceremony, and although the amount of documentation produced in Agile development may be much less than that which is produced using a process such as RUP, Agile does have clearly defined and rigidly adhered to processes.

One of the most well-known examples of an Agile process is Extreme Programming, known as *XP*. XP is in stark contrast to the Waterfall process model, because it advocates very short development and release cycles, parallel work on different activities, very low levels of ceremony, and concepts such as incremental design, which encourages designing for what is needed right now, rather than for what will be needed in the future[77].

Agile processes are really frameworks for the creation and customization of processes, for every project cycle and over multiple cycles. Thus the property of agility: a process found lacking is immediately adjusted.

[75] Messerschmidt, D., G., Szyperski, C. 2003. *Software Ecosystem* p. 82
[76] http://www.agilemanifesto.org
[77] Kroll, P., MacIsaac, B. 2006. *Agility and discipline made easy.* ch. 1

Agile processes emphasize a tight interaction with the user or customer, including operators, at all stages. Rapid prototyping is an important enabler of user inter-action, since user feedback can be gathered and assessed early and often. Rather than assuming that requirements can be captured ahead of time (or even at all), Agile processes assume that a dialogue with the user is maintained continuously throughout the entire software life cycle.

It is worth noting that Agile development approaches are often characterised as craftsmanship based. This characterisation is based on the fact that Agile processes favour people over process[78]. The issue of craftsmanship in software development is discussed in depth in the next chapter.

3.2.1.3.1 Scrum – a contemporary approach to Agile development

The scrum approach to software development is a contemporary technique that embraces the principles of Agile development. The main proponent of Scrum (Jeff Sutherland) is in fact a signatory to the Agile manifesto[79].

Scrum has been inspired by a development process which was first described by Hirotaka Takeuchi and Ikujiro Nonaka in a paper publish in Harvard Business Review in 1986[80].The approach to product development described by Takeuchi and Nonaka is a holistic process in the sense that the stages in the development process do not take place sequentially – in other words, the process is agile, and not a waterfall process. The approach described by Takeuchi and Nonaka is not limited to software development, and was conceived as a generic approach to all new product development. The name *scrum* was only attached to the process by Jeff Sutherland, but followed the rugby metaphor which the original authors proposed.

Nonaka and Takeuchi's *rugby* approach was derived from a study of new product development in a number of large companies: Fuji-Xerox, Canon, Honda and NEC[81]. They identified six characteristics of new product management development in each of these companies:

- **Built-in instability** – this is achieved by giving the development team a large amount of freedom. Other than a broad strategic goal, the team is left up to their

[78] McBreen, P. 2001. *Software Craftsmanship* p. 115
[79] http://www.agilemanifesto.org
[80] Nonaka, I. Takeuchi, H. 1986. *Harvard Business Review* p. 137-146
[81] Nonaka, I. Takeuchi, H. 1986. *Harvard Business Review* p. 138

own devices. An element of tension is created by giving the team both great freedom, but also great responsibility.

- **Self-organizing project teams –** because the team is given so much freedom, yet cannot rely on any prior knowledge, it must self-organize. This means that it functions like a start-up company by taking the initiative and the associated risks, and by developing an independent agenda. Nonaka and Takeuchi identify three criteria that identify a self-organizing project team: autonomy, self-transcendence and cross-fertilization[82]. *Self-transcendence* means that the team starts to set its own goals, which may exceed the original goals set to it by management. *Cross-fertilization* appears when a diverse group of people are employed in the team, and *knowledge transfers* take place seemingly effortlessly among the team members. Autonomy indicates that the team experiences very little day-to-day interference from senior management.
- **Overlapping development phases** – in the *rugby* approach, the development phases of a project are not isolated from each other. The phases overlap considerably, enabling *the group to absorb the vibration or noise generated throughout the development process*[83]. Even when a bottleneck appears, the process does not come to a halt, although the level of noise in the system may increase. Overlapping development phases necessitate a different approach to the division of labour than that which is used in the waterfall approach. This is because all team members are now expected to be actively involved in all phases – the delineation of responsibility is much less clear than in a waterfall approach. The process is more *intensive* to manage, since multiple phases with their concomitant issues have to be dealt with simultaneously.
- **Multi-learning**. Team members are required to be aware of what all others are doing at all times, and to stay in touch with outside sources of information. The process requires trial and error. These factors combine to force the team members to obtain a broad level of knowledge and skill. Nonaka and Takeuchi identifies the learning as taking place across two dimensions, namely across multiple levels, and across multiple functions[84]. Multi-level learning is learning that takes place at

[82] Nonaka, I. Takeuchi, H. 1986. *Harvard Business Review* p. 139
[83] Nonaka, I. Takeuchi, H. 1986. *Harvard Business Review* p. 141
[84] Nonaka, I. Takeuchi, H. 1986. *Harvard Business Review* p. 141

different group and aggregation levels in the organisation. These levels of organisation stretch from learning at the individual level and learning at the level of the product development group to learning at the level of the organisation itself. Multi-function learning is learning that takes place outside the area of knowledge or expertise an individual has.

- **Subtle Control**. The scrum approach advocates giving project teams a large degree of freedom and leaving a great deal of initiative up to the team. Nevertheless the teams are not uncontrolled. *Management establishes enough checkpoints to prevent instability, ambiguity, and tension from turning into chaos*.[85] This control is established in a variety of ways, some of which include careful selection of team members, tolerating and anticipating mistakes, and establishing evaluation and reward systems based on performance.
- **Organisational transfer of learning**. Not only does learning take place across multiple levels and functions within the product development team, but it also takes place amongst different product development teams. This is encouraged by, for example, placing members of one team into other teams.

Nonaka and Takeuchi saw scrum as a way for organisations to adapt to ongoing changes in the marketplace which forces organisations to be ever more flexible and dynamic[86].

Jeff Sutherland developed the concept of scrum into a more formalised approach to product development. Scrum is particularly used in software development, with in excess of two thousand professionals having been certified as Scrum Masters in 2005[87]. The first attempt to conduct a development project using the principles of scrum that Nonaka and Takeuchi had identified took place at Easel Corporation in 1993[88]. The project was successful, and demonstrated that the techniques of scrum can be used to formulate a viable software development strategy.

Jeff Sutherland continues to champion scrum as a technique for software development. In a 2005 lecture delivered to the Agile 2005 Conference, in Denver, Colorado, he addressed the future of scrum as a software development approach. His proposals rely on the way in which the phases of development overlap during the scrum process –

[85] Nonaka, I. Takeuchi, H. 1986. *Harvard Business Review* p. 143
[86] Nonaka, I. Takeuchi, H. 1986. *Harvard Business Review* p. 146
[87] Sutherland, J. 2005, *Agile 2005 Conference* p. 1
[88] Sutherland, J. 2004. *Cutter Agile Project Management Advisory Service: Executive Update* p. 1

these are shown in the figure below.

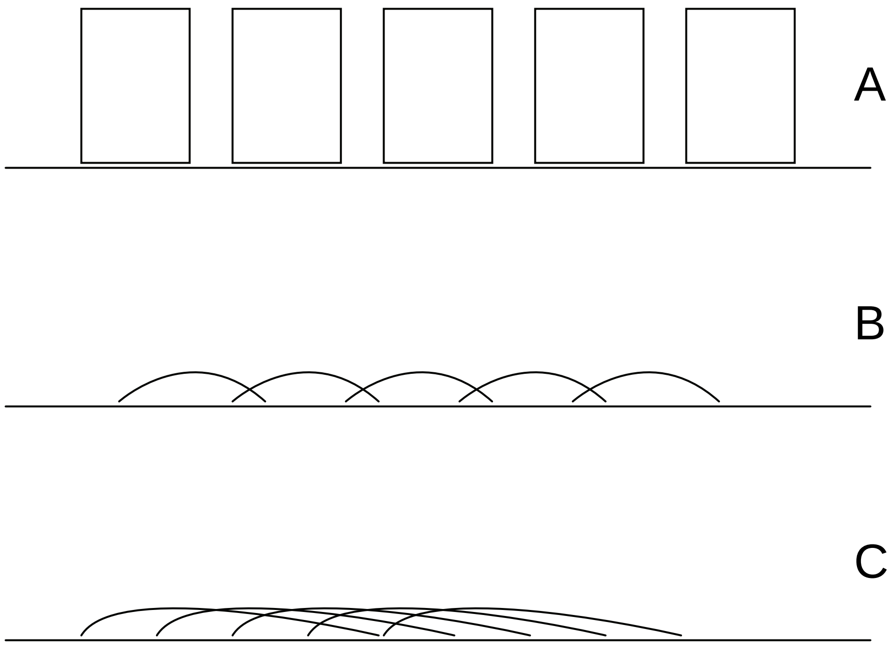


Figure 3-4: Sequential (A) versus overlapping (B and C) phases of development
Source: Adapted from Nonaka and Takeuchi 1986

The type A development process is a waterfall process - development in silo, or as a relay race where the results of each concluded phase is handed over to the next phase. Type B shows a process where the phases overlap at least in the beginning and the end. Type C is a process where the overlap may span more than one phase. Sutherland proposed that the abstraction level represented in the diagrams can be elevated, so that scrum itself can be seen as being of one of the three types A, B or C. Rather than regarding types B and C as scrum, and type A as not-scrum, he regards *traditional* scrum as type A. *Traditional* scrum is where units of work are contained within an iteration of work called a *sprint*. Each sprint completes, and then the next sprint is started. According to Sutherland this leads to lessened productivity[89], because time is lost in planning the next sprint before starting it. His proposal is to include product definition tasks into each sprint, leading to an overlapping set of sprints, such as in B. By taking this one step further, and letting one scrum team do software releases at the

[89] Productivity in this research, refers to *a performance measure that indicates how effectively an organization converts its resources into its desired products or services*. (O'Neil, S.L, Hansen, J.W. 2001. *Encyclopedia of Business and Finance* p. 708-710)

same time as software development, one reaches the type C scrum approach[90]. Sutherland acknowledges that it requires an experienced scrum team to be able to take full advantage of a Type C approach. The figure below shows the evolution of scrum that Sutherland envisions. In this vision a *meta-scrum* exists within which various smaller sprints are subsumed..

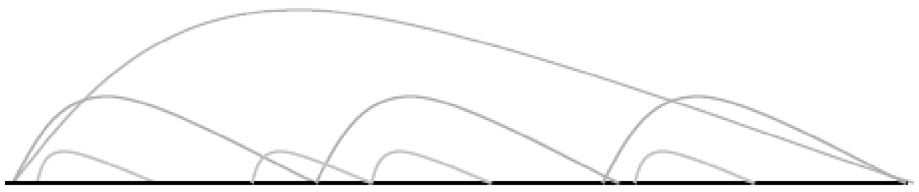


Figure3.5: Simultaneous overlapping Sprints running through a single set of development teams

Source: Sutherland, 2005, page 6

Sutherland says that the Type C scrum may offer an overwhelming competitive advantage to organisations that implement it, due to vastly increased speed of development, alignment of individual and corporate objectives, the creation of a performance driven culture, shareholder value creation, stable and consistent communication of performance, and the enhancement of individual's development and quality of life[91].

Agile development in the form of scrum is gaining ground, based on the ability of the technique to cope with a changing and fluctuating environment.

The preceding sections presented an overview of the most common approaches to the process of software development. The next sections describe various approaches to the design of software – design constitutes a phase in the software development life cycle, but the approach to software design has a bearing on the production of software, which in turn reflects on the process of industrialisation.

3.2.2 Design Paradigms

Design is the activity of transforming requirements specifications into a technically feasible solution.[92] This brief definition describes a core activity in the software

[90] Sutherland, J. 2005, *Agile 2005 Conference* p. 2-10
[91] Sutherland, J. 2005, *Agile 2005 Conference* p. 10
[92] Albin, S., T. 2003. *The art of software architecture* p. 25

development life cycle – the design process is an integral part of both the software engineering and the architectural views on the software development life cycle.

The most common approaches to software design are either the algorithmic approach, or the object-oriented approach[93]. The algorithmic approach to software development focuses on a decomposition of software into functions and procedures. This tends to lead to *brittle* systems, meaning that the software is not resilient in the face of change – as requirements change, it becomes extremely difficult to incorporate changes into the system without having to fundamentally rewrite the software[94].

Modern enterprise systems are most often designed using an object-oriented approach to system design. This approach uses abstraction as the main driver of design and modelling, and achieves abstraction by encapsulation. This is discussed in more detail in the section Object oriented design, below.

New design paradigms, which have evolved from the object-orientation paradigm, have recently been coming to the fore. An overview of these design paradigms are given in the section After objection orientation.

3.2.2.1 Object oriented design

Greenfield[95] states that *dealing with complexity was the primary motivation for most of the features that collectively form the paradigm of object orientation*. The primary means that the paradigm uses to address issues of complexity is encapsulation. According to Greenfield, *encapsulation raises the level of abstraction by wrapping one or more related data elements in a software construct*. Encapsulation raises the level of abstraction in design – by using objects that encapsulate a number of concepts, it is not necessary to understand the implementation details of the encapsulated entities.

The basic building block of object-orientation is the concept of a *class*, coupled with the related idea of an *object*. A class is defined as: *a description of a set of objects that share the same attributes, operations, relationships, and semantics*[96]. A class is therefore a description of a type of entity.

An object, in turn, is defined as *a concrete manifestation of an abstraction; an entity*

[93] Booch, G., Rumbaugh, J., Jacobson, I. 1999. *The unified modeling language user guide* p. 10
[94] Booch, G., Rumbaugh, J., Jacobson, I. 1999. *The unified modeling language user guide* p. 11
[95] Greenfield, J. Short, K. 2004. *Software Factories* p. 66
[96] Booch, G., Rumbaugh, J., Jacobson, I. 1999. *The unified modeling language user guide* p. 459

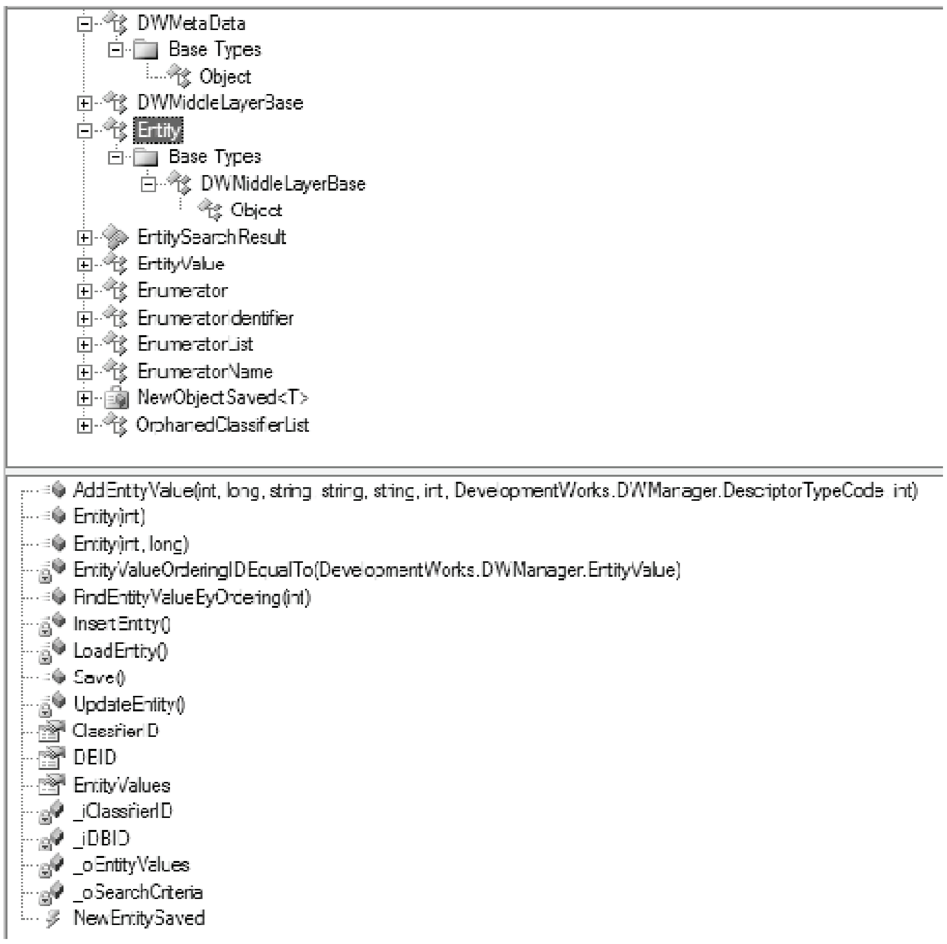
with a well-defined boundary and identity that encapsulates state and behaviour; an instance of a class[97].

An object represents an entity, and a class a description of an entity.

The object-orientation paradigm is supported at various levels during the software development process, as enumerated below:

- **Programming languages**: Various programming languages support object-oriented syntax to a larger or lesser degree. Languages such as C# and C++ support the main concepts of object-orientated programming (i.e. inheritance and polymorphism). Languages such as Ruby and Smalltalk embrace the concepts of object-orientation to the extent that every entity within the language is itself an object. Wikipedia provides a list of programming languages that either support object-orientation features, or are completely compliant with the ideas of object orientation[98].
- **Operating Systems**: entities within the operating system are presented as objects – for example, when administering user rights in a system, the *user* is an object, and one sets the properties of the object to change the user’s rights.
- **Development tools**: the user interfaces that development tools provide to users, support the concepts of object orientation, by presenting views on the code that shows the code as objects. The tools themselves are frequently written in object-oriented languages such as C++ or Java. A view of an object-oriented application as produced by Microsoft's Visual Studio is shown below. The view is generated by the development tool, and is based on the source code that the developer is creating. The upper part of the view shows the entire set of classes that has been created in the application, and the lower part of the view shows the details of the class that has been selected in the upper part of the view. The ability to obtain this kind of graphical overview of a program in the development environment encourages developers to adopt object oriented design paradigms in order to take advantage of the tools.

[97] Booch, G., Rumbaugh, J., Jacobson, I. 1999. *The unified modeling language user guide* p. 464
[98] http://en.wikipedia.org/wiki/Object-oriented_programming_language



```
DWMetaData
  Base Types
    Object
DWMiddleLayerBase
Entity
  Base Types
    DWMiddleLayerBase
      Object
EntitySearchResult
EntityValue
Enumerator
EnumeratorIdentifier
EnumeratorList
EnumeratorName
NewObjectSaved<T>
OrphanedClassifierList
```

```
AddEntityValue(int, long, string, string, string, int, DevelopmentWorks.DWManager.DescriptorTypeCode, int)
Entity(int)
Entity(int, long)
EntityValueOrderingIDEqualTo(DevelopmentWorks.DWManager.EntityValue)
FindEntityValueByOrdering(int)
InsertEntity()
LoadEntity()
Save()
UpdateEntity()
ClassifierID
DEID
EntityValues
_iClassifierID
_iDBID
_oEntityValues
_oSearchCriteria
NewEntitySaved
```

Figure 3.6: Class view in Microsoft Visual Studio

Source: Author

The need to visualise and document object-oriented systems led to the specification of the widely adopted Unified Modelling Language, commonly known as the UML. The UML was developed as a unified version of a number of extant modelling languages. By means of a process of public participation, and managed by a core team, the first version of the UML was accepted as a standard in 1997[99]. The UML is a language for visualising and specifying object-oriented models[100].

Closely associated with object-oriented design, is the concept of *Design Patterns*. The concept of design patterns was formulated by Gamma, Helm, Johnson and Vlissides in a book first published in 1994. Design patterns are *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular*

[99] Booch, G., Rumbaugh, J., Jacobson, I. 1999. *The unified modeling language user guide* p. xx
[100] Booch, G., Rumbaugh, J., Jacobson, I. 1999. *The unified modeling language user guide* p. 15

context[101].

Object-oriented design forms the basis of component based development. The latter extends object-oriented development by going beyond encapsulation, and *building on the idea that structure and behaviour can be formalized and analyzed in terms of component interactions*[102]. A component is a *physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces*.[103]

Object-oriented design principles are very widely followed in modern software development, mainly because they have been proven to be of value in developing software in a large number of problem domains, and at varying levels of complexity and size. Despite the utility of the object-oriented design approach, Sircar[104] et al concludes that this approach is not a revolutionary innovation, but should rather be seen as an evolutionary step towards the goal of industrialised software production.

3.2.2.2 After object orientation

Object orientated design has provided a solid basis for future innovation in software design paradigms, but the paradigm does not address all the issues with software development. Objects provide a level of abstraction, but are too fine-grained to model very large systems.

Greenfield lists the following chronic problems of software development, which cannot be solved by the object-orientation development paradigm[105]. He identified the problems listed here by looking at the issues that face software development projects, despite the fact that they adopted an object-oriented approach. The issues are:

- **Monolithic Construction**. Despite the promise of object-orientation, component re-use does not take place on a commercially significant scale, and virtually all software applications are written from scratch.
- **Gratuitous Generality**. Greenfield's contention is that the level of generalisation offered by object-orientation and its associated tools (i.e. UML) is too wide – the level of abstraction should be raised further, in order to provide a more focused development paradigm.

[101] Gamma, R., et al. 1995. *Design Patterns* p. 3
[102] Greenfield, J. Short, K. 2004. *Software Factories* p. 70
[103] Booch, G., Rumbaugh, J., Jacobson, I. 1999. *The unified modeling language user guide* p. 459
[104] Sircar, S. Nerur, S., Mahapatra, R. 2001. *Revolution or Evolution ?*
[105] Greenfield, J. Short, K. 2004. *Software Factories* ch. 4

- **One-Off Development**. Software is generally developed to cater for a specific set of requirements, meaning that every piece of software has to be developed from the beginning to cater for new requirements. The concept of once-off development is fostered by the nature of processes such as the Unified Process and Agile, which are reliant on requirements.
- **Process Immaturity**. The management of software development processes are notoriously difficult, and projects often end in failure, due to budget overruns or functional incompleteness.

Greenfield et al identifies a number of critical innovations that may address the issues object-orientation cannot address.

- **Systematic Reuse**. Greenfield identifies the development of *program families* as the primary innovation that will drive systematic reuse. A program family is a group of software products of which the members are different, but share many common features.
- **Development by Assembly**. This refers to the ability to assemble independently developed components across platform boundaries with predictable results. A number of technological innovations are making this a feasible scenario, including the development of platform independent protocols and self-descriptive components.
- **Model Driven Development**. This refers to raising the level of abstraction for developers above that provided by object-oriented design. The level of abstraction is achieved by effectively developing specialized languages to describe models in different domains. The model can then be used as the abstraction.

The innovations described by Greenfield et al, form the basis for the concept of *Software Factories*. This concept may lay the foundations for the industrialization of the software development industry. The details of software factories and the process of industrialization is the subject of the following chapters.

3.3 Conclusion

This chapter provided an overview of the current state and tools used in the development of enterprise software. The intention of the chapter was to sketch the domain of software development, and to illustrate that software development is an inherently difficult process, due to the essential properties of software development as identified by Brooks.

The actual process of software development is encapsulated in the software development life cycle. Various approaches to this, expressed as process models were discussed.

At the time of writing the Agile approach to software development, as exemplified by Sutherland's Scrum technique, has been gaining a great deal of traction in the software development community. The main reason for this is that Agile techniques, and Scrum specifically, is well suited to being deployed in a dynamic and changing environment, which is the environment in which many organisations currently find themselves.

It was pointed out that in order to achieve an overarching view of the software development process it is necessary to adapt different views, which complement each other to give the complete picture.

The development process itself relies on the design paradigm that is introduced into the development process by the software architect. At the time of writing, this paradigm is very often that of object-orientation. The object-orientation paradigm does not, however, solve all of the difficulties of the software development process, as can be seen in the enumeration of chronic problems which crop up during the development process.

A number of recent innovations, which collectively constitute *Software Factories*, are part of an attempt to address the chronic issues mentioned above. In particular, these innovations may form the foundation of an industrialized approach to software development. The industrialization of the software development industry is the subject of the next chapter.

Chapter 4 Industrialisation and the Economics of Software Production

This chapter provides insight into the concepts of industrialisation, as it applies to the software industry[106]. The software industry does not create a tangible, physical output, and the contention of this chapter is that the industrialisation of this discipline will not manifest itself as an increase in the absolute output of software, but rather as an increase in the quality and re-usability of the products of the software engineering discipline. The achievement of large-scale economies of scope, rather than economies of scale, will be the hallmark of industrialised software production.

Specific mechanisms that may facilitate and encourage the move towards the industrialisation of software development is presented in the shape of software factories and cloud computing. Software factories represent a possible route towards industrialisation, and cloud computing is a rising paradigm for information technology service delivery that may encourage the process of industrialisation.

4.1 Industrialisation

Industrialisation, as a concept, is used to describe an evolving process of economic and social change that leads to vastly improved manufacturing efficiencies in a specific industry. The process of industrialisation manifests itself in different ways in different industries. The many different expressions of industrialisation are in part due to the fundamental differences amongst sectors of industry, and to the variety of products each sector produces. The extent of diversity amongst industrial sectors is illustrated by the North American Industrial Classification System (NAICS), which lists 1170 codes for industries[107]. This means that there are 1170 recognised, differentiated industrial sectors in the United States.

[106] The phrase *industry* refers to *manufacturing or technically productive enterprises in a particular field* [Dictionary.com], and does not imply that the sector is *industrialised*.
[107] Kaliski, B., S. 2001. *Encyclopedia of Business and Finance (Vol 2)* p. 642

While industrialisation may mean different things in different industries, some of the typical manifestations of industrialisation within an industry, are the following[108]:

- The industry customises and assembles standard components, to produce similar but distinct products.
- Production processes are standardised, integrated and automated.
- The industry has developed or is developing extensible tools, and is configuring these tools to automate repetitive tasks.
- Customer and supplier relationships are being leveraged to reduce risks and costs.
- The production of product variants is automated using product lines.
- Production is distributed across supply chains, which consist of specialised (and interdependent) suppliers.

Regardless of the industry, industrialisation represents a shift from skills to technology. The invention of *technology* in the 18th century led to the so-called industrial revolution. According to Drucker[109] the roots of the word technology reflects the essence of industrialisation: *techne* (the mystery of a craft or skill) combined with *logy*, indicating organised, systematic and purposeful knowledge. The *Encyclopédie, ou dictionnaire raisonné des sciences, des arts et des métiers* (In English - *Encyclopedia, or a systematic dictionary of the sciences, arts, and crafts*) is, according to Drucker, one of the most important documents originating from the period of the industrial revolution. The *Encyclopédie* was published in France, starting in 1751, with additional volumes being published periodically up to 1772 when the final plates were published[110].

The *Encyclopédie* consists of a large number of articles that detail the knowledge necessary to master a very large number of arts and crafts, with the content presented in a systematic and ordered way. The underlying thesis of the *Encyclopédie* was that *effective results in the material universe – in tools, processes, and product – are produced by systematic analysis, and by the systematic, purposeful application of knowledge*[111].

The codified knowledge in the *Encyclopédie* represented the first significant body of

[108] Greenfield, J. Short, K. 2004. *Software Factories* p. 156
[109] Drucker, P., F. 1993. *Post-capitalist society* p. 28
[110] http://www.lib.uchicago.edu/efts/ARTFL/projects/encyc/volumes.html
[111] Drucker, P., F. 1993. *Post-capitalist society* p. 28

knowledge that codified and published the mystery of craft - this codification was essential to the industrial revolution. The industrial revolution utilised codified knowledge to attain greater manufacturing efficiencies. The consequences of the revolution included the coming into being of factories – since knowledge could not effectively be applied in many small loci of production, the factory was born out of the necessity to concentrate production. Factories in turn require large energy sources. The development of factory-based manufacturing and the supply of energy to these centres of manufacturing, contributed to the rapid shift from craft-based production to technology-based production.[112] This process, the process of industrialisation, had an impact on both the social and economic environment.

The specifics of industrialisation differ amongst differing industries, but are an ongoing process. New industries are not able to immediately take advantage of the manufacturing and production efficiencies that can be attained by industrialisation of the industry. An example of a specific industrialisation process is that surrounding the industrialisation of the microcomputer[113] industry, (referring to hardware, not software). A thorough overview of the industrialisation process during the inception phases of the micro computer industry is presented by Langlois, in a 1992 paper entitled *External Economies and Economic Progress: The Case of the Microcomputer Industry*[114]. The paper clearly reflects the trend towards standardisation in the industry, as described below.

The earliest microcomputers were hand-build, and used primarily by a group of enthusiasts known as the *Homebrew Computer Club*, located in California in the United States. This group of enthusiasts together with the engineers and hobbyists who built the first prototype microcomputers, fulfilled the role of craftsmen in the history of the microcomputer. With technological innovation, microcomputers or Personal Computers (PC's) were produced to meet the requirements of the mass consumer market. The process of production moved from the manual assembly of the first microcomputers

[112] Drucker, P., F. 1993. *Post-capitalist society* p.29

[113] A microcomputer is distinct from a mainframe or a minicomputer. The distinction is made on the basis of the number of concurrent users the machines can support, and to an extent the peripheral support systems and personnel required to maintain the systems. Minicomputers are personal-use machines, with a very low cost compared to other systems. Minicomputers are not widely used currently - the distinction between minicomputers and microcomputers are becoming increasingly blurred over time. Mainframes can support very large numbers of concurrent users, require dedicated infrastructure, and are typically used by large corporations for bulk data processing. Mainframes occupy the high end of the price range.

[114] Langlois, R., N. 1992. *The Business History Review*

(the Altair 8800), to the streamlined manufacturing processes that are in use today, where the components for personal computers are created by a large number of independent manufacturers, and only the final products are assembled by the brand vendors. The microcomputer industry has evolved from a cottage industry to a highly-industrialised industry in a period of less than three decades.

The microcomputer industry, as described above, can be categorised as being part of an industry that produces secondary, or derivative products. There are fundamental differences amongst the outputs of different industrial sectors. This is reflected in the large number of industrial sectors mentioned above, and also in the different sectors of economic activity to which each industry belongs. The actual number of sectors of economic activity number at least three[115]: the primary, which is concerned with the transformation of raw resources; the secondary, which is concerned with the production of derivative products, in other words manufacturing; and the tertiary sector, which involves service industries. Each economic sector has a different class of output, and within these output classes, each industry produces a vast number of products.

Because of the range and variety of output amongst industries, industrialisation cannot be recognised by examining specific processes within industries, but can be recognised by noticing changes in the production or manufacturing process that lead to greater productivity or manufacturing efficiencies, usually due to the appearance of economies of scale or scope through the application of technology. Economies of scope and scale are examined in the next section.

4.2 Economies of Scope and Scale

Economies of scale and scope both result in reduced time and cost and improvements in product quality by *producing multiple products collectively*, but they are very different in terms of the mechanism through which they are achieved.[116]

Economies of scale arise from the scale of the enterprise, while economies of scope arise from the scope of the enterprise. The scale of the enterprise refers to the size of the output, while scope refers to the variety of products being produced.

Economies of scope are a more modern concept than economies of scale, the latter

[115] Wolfe, M. 1955. *The Quarterly Journal of Economics* p. 406
[116] Greenfield, J. Short, K. 2004. *Software Factories* p. 157

already being mentioned by pioneering economists such as Adam Smith[117]. Economies of scale *are reductions in the average costs attributable to production volume increases*[118]. Economies of scale arise when *multiple identical instances of a single design is produced collectively, rather than individually*[119].

This type of economy appears when production volume is increased, leading to a reduction in the average price of production. Economies of scale can be classified as either internal or external economies. Internal economies of scale appear when a firm reduces costs by increasing production, as defined above. External economies of scale appear when an entire industry benefits from expansion. Koch and Inman cite the creation of an improved transportation system, a skilled labour force, or the sharing of technology as samples of expansion that will lead to external economies of scale.

There are a number of reasons why economies of scale may occur:[120]

- **Specialisation**: in a large enterprise, a specialised workforce, or specialised machinery, may be more efficient at performing tasks than machines or personnel which have to perform multiple tasks.
- **Volume discounts**: by performing at high output levels, a firm may be able to take advantage of volume discounts in purchasing input materials – suppliers will often be willing to supply large quantities at discounted prices.
- **Economic use of by-products**: By-products can often be economically exploited in the presence of large-scale production. Large oil firms produce numerous petroleum by-products. A further classic example is that of the meat industry, which produces leather as a by-product.
- **Market advantage**: large firms may find themselves in a position to manipulate the market, or even establish monopolies, leading to an ability to determine pricing at highly profitable levels.

Economies of scope were first discussed by Baumol, Panzer and Willig in a volume published in 1982, titled *Contestable Markets and the Theory of Industry Structure*. Building on this, later authors have defined economies of scope to exist if *a firm can produce several product lines at a given output level more cheaply than a combination*

[117] Langlois, R., N. 1992. *The Business History Review* p. 3
[118] Koch, J., C., Inman, A., R. 2006. *Encyclopedia of Management 5th ed* p. 209
[119] Greenfield, J. Short, K. 2004. *Software Factories* p. 157
[120] Koch, J., C., Inman, A., R. 2006. *Encyclopedia of Management 5th ed* p. 209- 211

of separate firms each producing a single product at the same output level.[121] In contrast with economies of scale, the cost advantage accrues to an organisation due to its ability to produce a (complementary) variety of products, rather than due to large production volumes. An example of economies of scope are the various services offered by banks – they offer related services such as retail banking and investment services through the same service infrastructure.

Greenfield[122] et al define economies of scope as *reducing the cost of solving multiple similar but distinct problems in a given domain by collectively solving their common subproblems and then assembling, adapting, and configuring the resulting partial solutions to solve the top-level problems*. This latter definition is much more specific and detailed than the definitions given in the general economic literature, because it details the mechanism via which economies of scope is achieved in an industry. Since economy of scope is not defined in terms of the specific mechanism, but rather in the terms of the general principle of savings due to the ability to take advantage of scope, this definition can be used as a valid approach to interpreting economies of scope, in particular as they apply to the software industry.

The next section examines the state of practice in software engineering, prior to linking the process of industrialisation to the software development industry.

4.3 The State of Practice in Software Engineering

This section examines the notion of software as a craft, and contrasts the software industry to an industrialised manufacturing process, such as that used to produce microprocessors.

The first international conference on software engineering was held in Garmisch, in Germany, during the course of 1968[123]under the auspices of the North Atlantic Treaty Organisation. The conference was hailed as being the first occasion on which the crisis in the way that software is being developed was openly discussed. The following quote from an invited talk by Doug McIllroy during the Garmisch conference summarises the problems that were being perceived at that time:

We undoubtedly produce software by backward techniques. We undoubtedly get the

[121] Koch, J., C., Inman, A., R. 2006. *Encyclopedia of Management 5th ed* p. 211
[122] Greenfield, J. Short, K. 2004. *Software Factories* p. 159
[123] Randell, B. 1979. *Proceedings of the 4th International Conference on Software Engineering* p. 8

short end of the stick in confrontations with the hardware people because they are the industrialists and we are the crafters. Software production today appears in the scale of industrialization somewhere below the more backward construction industries. I think its place is considerably higher, and would like to investigate the prospects for mass-production techniques in software[124].

This quote is from a talk given at a conference thirty nine years ago, at the time of this writing[125]. Four decades ago, the leading professionals in the software industry were in agreement that software development at the time bore more resemblance to a craft than to an industrialised process. Eischen notes, however, that the follow-up to the 1968 conference, which was held in Rome in 1969, once again under the auspices of NATO[126], and which was meant to address technical issues from the software engineering field, failed to produce an agreement on how to reconcile the practise and theory of software development. Software engineers seemed to agree that a problem existed in the industry, but could not agree on the approaches needed to solve the problem[127]

The problems in the software industry have persisted at least into the middle of the 1990's. An article appearing in *Scientific American*, in 1994, highlighted the issues besetting the industry, and made reference to the 1968 Garmisch conference as the point when the industry realised that there existed severe problems with the way software was being produced[128]. The 1994 article lists a number of software project failures, and states that the goal of software engineering is nothing but a *term of aspiration* in 1994[129]. The article notes a number of areas of progress, but also areas of concern within the state of the software industry, at the time of its writing:

- **Processes** – in 1994, processes such as the Capability Maturity Model made it possible to measure the level of ability of a team to meet the demands of a project.
- **Error reduction** – the application of stringent error testing, and mathematical modelling of execution paths in critical software systems, had shown that the number of errors in a product can be reduced significantly.

[124] Naur, P., Randell, B. 1969. *Software Engineering: Report on a Conference...* p. 10
[125] Naur, P., Randell, B. 1969. *Software Engineering: Report on a Conference...* p. 9
[126] Buxton, J., N., Randell, B. 1970. *Software Engineering Techniques*
[127] Eischen, K. 2002. *Computer* p. 37
[128] Gibbs, W., W. 1994. *Scientific American* p. 86
[129] Gibbs, W., W. 1994. *Scientific American* p. 87

- **Productivity measurement** – to a large degree, the industry did not measure productivity, and had no means of determining how productive a particular programmer was.
- **Professionalism –** at the time, it was recognised that software was still produced by craftsmen, rather than professionals. There was a dearth of training and professional certification options for software engineers at the time.

In a more recent overview of the state of practice in the software industry, the IEEE Software publication published a series of articles in November 2003, attempting to illuminate the issue. They identified a number of different trends, each of which is briefly discussed below:

- **Internet development –** development for internet, or web applications, differ from traditional desktop based applications in that the applications tend to be exposed to a much larger audience, and that the rate of change in this particular domain is very high. Most enterprise applications expose their user interfaces via the web, and makes use of the same technologies employed in non-enterprise web applications, such as social networks. The article noted that development practices for *internet speed development* were different from that of more *traditional* software applications – in particular the development was more agile, and less reliant on formalised procedures. Speed is of the paramount importance in this type of development, and much less ceremony is required during the development process. The authors regard internet development to be a completely new development paradigm.[130]
- **Systematic reuse** – systematic reuse is exemplified by the creation of software product lines.[131] The authors observed an industry wide trend toward implementing

[130] Baskerville, R., et al. 2003. *IEEE Software 20(6)* p. 70 - 77
[131] Greenfield describes a software product line as a mechanism to exploit the observation that it is easy to build software components that are reusable in the context of a particular domain or family of systems, rather than components that are generically reusable. According to Greenfield, page 125: *A software product line exploits these observations, identifying the common features and recurring forms of variation in a specific domain to produce a family of software products more quickly, more cheaply and with less risk and higher quality than would be feasible by producing them individually. Rather than hope naively for ad hoc reuse opportunities to serendipitously arise under arbitrary circumstances, a software product line systematically captures knowledge of how to produce the family members, makes it available in the form of reusable assets, such as components, processes, and tools, and then applies those assets to produce the family members. Products developed as the members of a family reuse requirements, architectures, frameworks, components, tests and many other assets.*

> software production lines in order to gain competitive advantage. The creation of a software product line is seen by many organisations as imparting a strategic advantage. The authors finally observe that software product lines are not yet a mature approach to software engineering.[132]

A third article appearing in the November 2003 edition of the IEEE Software publication pointed out the differences in software development practises in different countries, with very clear differences in approach amongst countries such as India and the United States emerging from the report.[133]

The current state of the software industry (2008 at the time of writing) has evolved since the IEEE Software publication of 2003. Of particular interest to enterprise software development is the emergence of the paradigm of software and services, and the utilisation of internet technologies to facilitate the interoperability and distribution of components. Software and services provide an alternative architectural approach to building enterprise solutions[134], and support the aim of systemic reuse and development by assembly that is expounded by Greenfield.[135]

From the discussion it appears that the software industry is emphasizing reusability, and various architectural patterns are emerging to support this trend. Software development paradigms are continuously adjusting to the environment and to accommodate new technological innovations. This is demonstrated by the observations made regarding different approaches to software in the internet domain of software application development.

There is as yet no unified approach to addressing the problems in the industry which had been identified in 1968[136] – there still exists fundamental differences of opinion as to whether software is (and can only ever be) a craft, or whether industrialisation is possible and inevitable.

The software industry appears to have been in a crisis for at least the past four decades as shown by the preceding discussion. The discussion has highlighted that this crisis has been precipitated by the lack of an engineering discipline being applied to the software

[132] Birk, A., et al. 2003. *IEEE Software 20(6)* p. 52 - 60
[133] Cusumano, M. 2003. *IEEE Software 20(6)* p. 28- 34
[134] Sangwell, K. 2007. *The Architecture Journal, Journal 13* p. 18-23
[135] Greenfield, J. Short, K. 2004. *Software Factories* p. 125 - 129
[136] Naur, P., Randell, B. 1969. *Software Engineering: Report on a Conference...*

industry. The lack of an applied engineering discipline is closely related to the lack of an industrialised process of software production.

4.4 Software as Craftsmanship

The previous section highlighted the state of practise in software engineering. It was shown that members of the industry have been aware for a long time that there is a degree of craftsmanship inherent in the practise of software engineering.

The idea that software development is naturally suited to a craft-based approach, and not at all to an engineering approach, has been put forward by a number of authors. A comprehensive argument for the case that an engineering approach is misplaced as a management tool for software development is given in McBreen's book *Software Craftsmanship*[137]. McBreen argues that the craftsmanship approach will allow organisations to grow the size of a core group of productive and efficient programmers beyond the size such a team normally assumes – there is a belief in the software development community that a small group of programmers produce the majority of the code[138]. Quoting McBreen:

> *Craftsmanship diverges from engineering in that it emphasizes personal responsibility and decentralization. Rather than supporting large training and accreditation organizations, software craftsmanship uses apprenticeship to make developers responsible for their own learning.*[139]

The view is supported by a number of other authors. Bill Pyritz of Lucent Labs[140] argues that establishing a journeyman system, whereby junior programmers are apprentices to master craftsmen, will lead to the development of a larger pool of skilled craftsmen in the long run – thereby enabling organisations to product higher quality software more efficiently (this directly supports McBreen's motivation for the craftsmanship based approach). Cockburn and Highsmith[141] likewise write in *Computer* in favour of this approach – they emphasize the central role people play in an Agile approach to software development. The authors come out in support of McBreen's thesis in *Software Craftsmanship*.

[137] McBreen, P. 2001. *Software Craftsmanship*
[138] McBreen, P. 2001. *Software Craftsmanship* p. 12
[139] McBreen , P. 2001. *Software Craftsmanship* p.179
[140] Pyritz, B., 2003. *Bell Labs Technical Journal 8(3)* p. 103-104.
[141] Cockburn, A., Highsmith, J. 2001. *Computer 34* p. 131-133

The experience of Borland Corporation in the early part of the 1990's, with the development of their Quattro Pro for Windows spreadsheet application, seems to bear out the view that the crafts-based approach may be a valid approach to application development[142]. The team employed by Borland consisted of a number of *highly productive professionals who viewed each other with the highest respect,* implying that the team members were chosen to be *master craftsmen*, in the sense that McBreen and Pyritz use the term. The success of the Quattro Pro project was reflected in the high productivity of the team members, and the high quality of the finished product.

It must be noted here that the general consensus amongst these authors is that the craftsmanship approach best applies to smaller development projects – nevertheless the Borland project was a very large scale project, albeit not on the order of typical telecommunications or similar projects.

Countering the opinions of the craftsmanship advocates is that of Phillip K Janert in his review of McBreen's work[143]. Janert points out that the apparently small percentage of people who are the most productive in software development (and which is used as a rationale for using the craftsmanship approach), is not a phenomenon unique to software development – the phenomenon may be seen to occur in various industries, some of which have move away from the purely craft based approach. Janert further states that the craftsmanship approach may be valid in certain environments, but may be neither the only nor the most optimal approach – software engineering does have its place, and it may be complemented by the craftsmanship approach.

Kyle Eischen sheds further light on the craftsmanship versus engineering debate in a article published in *Computer*, in 2002[144]. Eischen draws a parallel with the experience of numerous other industries over the past two centuries – many of these industries experienced great resistance to *industrialisation*, i.e. to moving away from the craftsmanship based approach, and it should therefore not come as any surprise that there is resistance within the software industry to a move towards industrialisation. Eischen states that the debate can be refocused by considering the rationalisation of the production of software. Eischen's approach to industrialisation is discussed in more detail in the next section.

142 Coplien, J.O. 1994. *Proceedings of the 5th annual Borland International Conference*
143 Janert, P.K. 2003. *IEEE Software 20(6)* p. 108-109
144 Eischen, K. 2002. *Computer 35(5)*

An interesting connection exists between the development of the scrum approach to software development, and the motivation for the belief that a crafts-based approach to software development is a feasible approach. Scrum was inspired partly by Borland's Quattro Pro development project[145]. As was pointed out above, the same project may be seen as a justification for the belief that a team of master craftsmen can build high quality software. The difference between the approaches is that the craftsmanship based emphasizes the skill of individuals, while the scrum approach emphasizes the skills of the team[146]. In the words of Sutherland: *The single-programmer model does not scale well for large projects.*

Both Nonaka's and Boisot's theories of knowledge creation support the notion of *diffusion*. In terms of Nonaka diffusion indicates that knowledge is being externalised – i.e. being made more explicit, and hence shareable. Boisot's theory concentrates on the notion of *diffusibility* - the least diffusible knowledge being tacit knowledge that is available only to the individual. The fact that a team can access knowledge indicates that the knowledge exists at a point on a scale of diffusibility where the knowledge can be shared. The theories of Nonaka and Boisot are discussed in detail in chapter five, below.

It is proposed that the retention of a craftsmanship based approach to certain aspects of software development may indeed enable organisations to grow the group of master craftsmen, but that this does not preclude the software development process as a whole from being industrialised. Although the possibility of industrialising the software development process is not the subject of this research, it should be noted that industrialisation on a larger scale is possible, indeed probable, as is discussed in the next sections.

4.5 Software as an Industry

Having considered software development as a craft, it is time to consider software production as an industry. The previous section made it clear that software production cannot yet be considered to be industrialised, and that there is a strong element of craftsmanship evident in the practise. The discussion in this section therefore assumes a future oriented viewpoint, asking whether the software industry can be industrialised,

[145] Sutherland, J. 2004. *Cutter Agile Project Management Advisory Service: Executive Update* p. 2
[146] Sutherland, J. 2004. *Cutter Agile Project Management Advisory Service: Executive Update* p. 2

and examining the possible routes to industrialisation.

Addressing the question of whether the software industry can be industrialised at all, the following quote by Kyle Eischen can be applied to the argument:

> *Arguably, the industrial revolution signalled the rise and dominance of rational approaches to production over small-scale, craft-based methods. In a world of scarce resources, producing more with less is a positive outcome.*[147]

Eischen puts forward the view that, although the software industry differs from other industries in certain respects, there is nothing in these differences that should lead one to believe that the industry cannot be industrialised, similar to the industrialisation process that took place during the industrial revolution for many other industries. There exists a perennial shortage of skills in the information technology sector – and it would be in the interest of the industry to maximise the output of these scarce skills.

Eischen identifies aspects of the software production process that sets it apart from other industries – it is, for example, hard to classify software products as products that should be patented, or should be copyrighted. In other words – software artefacts contain aspects of both creative and scientific endeavour. The author sees the eventual industrialisation of the software industry as inevitable, but postulates that the route to industrialisation may require a new approach to thinking about software development. The importance of domain knowledge and the difficulty of communicating effectively during the software production process may require new, cross-disciplinary training for software engineers in order to maximise the effectiveness of these scarce resources. The increasingly important role that software plays in our society further stresses the importance of raising the level of awareness of the industry to those not in the industry, and maximising the output of the people producing software. According to Eischen the industrialisation is inevitable, but will require that those playing a role in the software industry take wider cognisance of economic and social factors outside their domain.

Greenfield presents a different argument for the inevitability of the industrialisation of software production. According to Greenfield, the main force towards industrialisation is that *stakeholder expectations tend to grow as platform technology advances*.[148] He states that, up to the time of writing, the industry has been meeting the increased

[147] Eischen, K. 2000. *Computer 35(5)* p. 37
[148] Greenfield, J. Short, K. 2004. *Software Factories* p. 4

demand by honing the skills of the craftsmen, the *master programmers* mentioned by those advocating the craftsmanship approach. This solution is not scalable beyond a certain (as yet undetermined) point – because the demand will overwhelm the output since the potential output of a craftsmanship based approach is limited by the methods and tools, and the size of the available labour pool. This observation is based on historical facts gleaned from the industrialisation process in other industries.

Greenfield's core proposal is to facilitate the process of industrialisation that will be required to meet the demand, by *replacing apprenticeship with automation that exploits best practises*.[149] He acknowledges that it would be overly simplistic to suggest that software development could be reduced to a mechanical process, with little or no intervention. He does suggest that automation can help focusing the most highly skilled developers (the *master craftsmen* mentioned in McBreen's *Software Craftsmanship*) on tasks where they can be most productive.

In terms of the process of industrialisation, Greenfield notes that the comparison of the software industry to other industries is erroneous in the sense that the comparisons are often made on the basis of comparing the production of physical goods to that of software[150]. The industrialisation of software production will lead to the exploitation of economies of scope, rather than scale. The primary reason for economies of scope to arise is that the production of the physical artefacts embodying software is trivial, if not non-existing. Software is a logical asset, not a physical asset; therefore the greatest effort is concentrated in designing and customising the product. The design and customisation process includes the process of actually writing the software. Especially in the enterprise market, where very high levels of software customisation exists[151], it becomes impossible for economies of scale to arise – only economies of scope can exist. Even though the software industry differs from other industries in this particular respect, Greenfield reiterates that it is no different than any other industry in other respects – engineers develop designs and prototypes, and copies are mechanically produced to serve mass markets. As with every other industry, the software industry

[149] Greenfield, J. Short, K. 2004. *Software Factories* p. 5
[150] Greenfield, J. Short, K. 2004. *Software Factories* p. 156
[151] Large corporations utilise enterprise software to provide them with strategic advantages. To this end they will customise off-the-shelf solutions to suit their specific business models and operating procedures, or they will commission entirely customised solutions to cater for the vagaries of the business. In this sense enterprise software may be seen as a strategic asset to the organisation. This extreme degree of customisation is precisely the scenario that precludes economies of scale from arising.

experiences an imperative to maximise productivity, and minimise costs[152]. Software factories are the mechanism which Greenfield proposes to use to facilitate the industrialisation process. A brief overview of the concept of software factories is given below, as a practical example of a possible means of moving the software industry to the state of a fully-fledged industrialised undertaking.

In addition to software factories (which focus on the software production line) the consumer side of software industrialisation is represented by the concept of cloud computing. Cloud computing constitutes a mode of consumption of information technology services which will encourage the adoption of processes of industrialisation in software production. An overview cloud computing follows the overview of Greenfield's work.

4.5.1 Software Factories

Greenfield defines software factories as follows:

> *A software factory is a software product line that configures extensible tools, processes, and content using a software factory template based on a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework-based components*[153].

An analysis of the definition given above, leads us to notice the following salient points. First of all – a software factory is a factory in the sense that it constitutes a product line that uses standardised parts, tools and production processes even though it is not a factory that produces physical goods. The two central elements of a software factory, according to the definition, are the software factory schema and the software factory template (which is based on the schema). The factory schema defines the artefacts, and the assets used to build these artefacts[154]. The software factory template is the actual implementation of the artefacts described by the schema. By making use of domain specific languages, the software schema incorporates domain specific knowledge directly into the factory – each factory is build for a specific domain. Domain specific languages are a type of programming language (or a specification language) that is specialised to suit the needs of a particular domain[155]. A DSL models concepts found in

[152] Greenfield, J. Short, K. 2004. *Software Factories* p. 160
[153] Greenfield, J. Short, K. 2004. *Software Factories* p. 163
[154] Greenfield, J. Short, K. 2004. *Software Factories* p. 164
[155] Greenfield provides an extensive definition of the attributes of domain specific languages. (Greenfield,

a specific domain[156]. This is in contrast to general purpose languages – modelling languages such as UML, and programming languages such as C++ or COBOL. Domain specific languages (or DSL's) are so-called 4GL languages. This means that they provide a high level of abstraction of the underlying hardware and operating system.

The schema and the factory together enables one to build a software factory – this is in essence a product line that automates the design process, and removes much of the development burden from developers by employing the principle of systematic reuse.

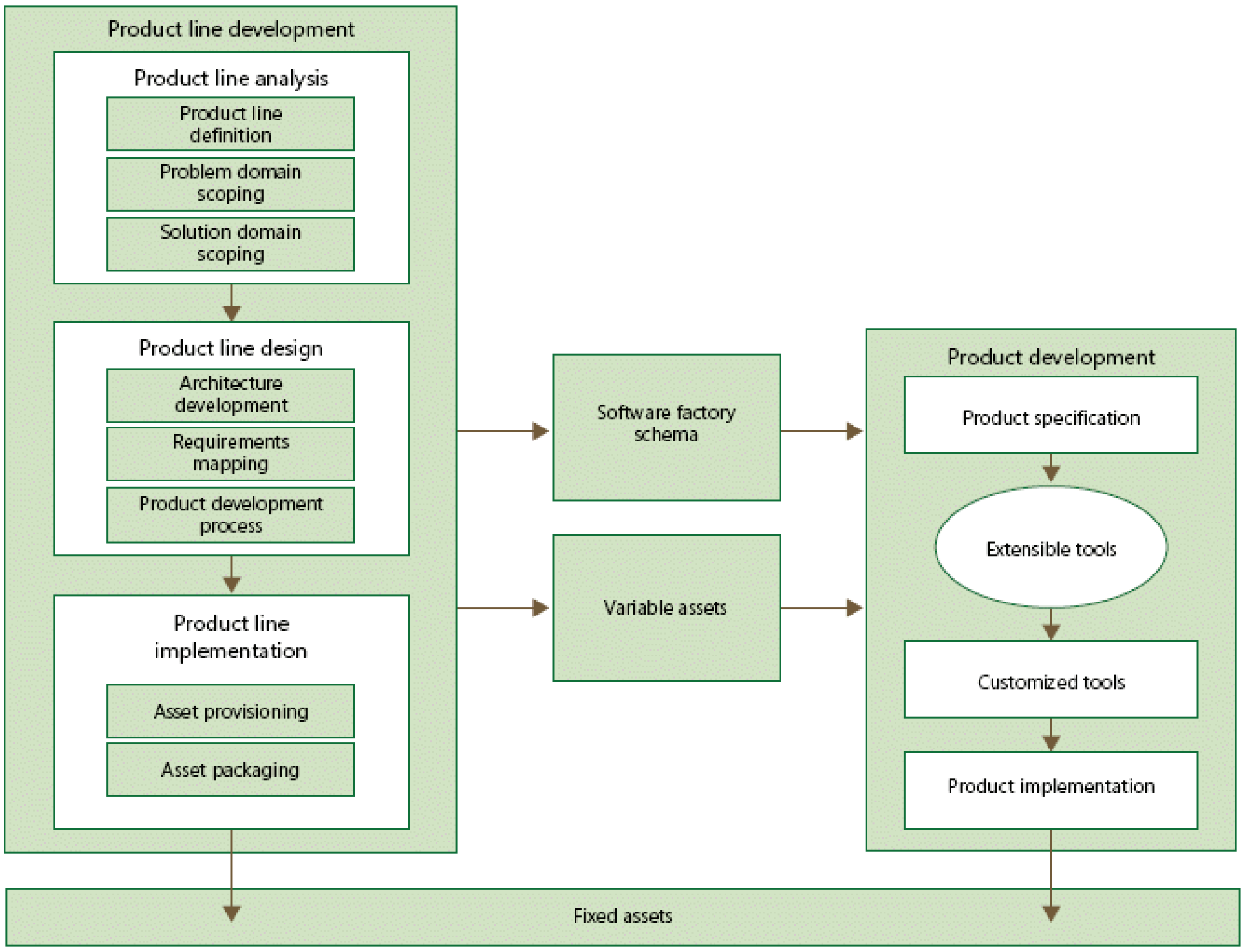


Figure 4.1: A software factory

Source: Greenfield, page 163

A more detailed definition of the terms software factory template and software factory schema are given below.

J. Short, K. 2004. *Software Factories* p. 142 - 142)

[156] Greenfield, J. Short, K. 2004. *Software Factories* p. 143

4.5.1.1 Software Factory Schemas

A software factory schema is a schema that *describes the artefacts that must be developed to produce a software product*[157]. This description may be expressed as a grid, or as a graph. The table below gives a sample grid, based on Greenfield[158].

	Business	Information	Application	Technology
Conceptual	- Use cases and scenarios - Business goals and objectives	- Business entities and relationships	- Business processes - Service factoring	- Service distribution - Quality of service strategy
Logical	- Workflow models - Role Definitions	- Message schemas and document specifications	- Service interactions - Service definitions - Object models	- Logical server types - Service mappings
Physical	- Process specification	- Database schemas - Data access strategies	- Detailed design - Technology dependent design	- Physical servers - Software installed - Network layout

The rows in the grid define levels of abstraction, and columns reflect the concerns of different stakeholders in the software product. By populating the grid with the development artefacts for a specific product, one takes the first step towards creating a software factory to produce this product.

Once the grid has been populated for a specific product, each cell becomes equivalent to a perspective, a viewpoint from which a certain aspect of the software development is seen. If the cells contain models, then the models are described in domain specific languages, once again reinforcing the fact that the schema applies to a specific product. The viewpoints represented by the cells describe the languages that will be used to develop the artefacts as well as the constraints in the form of requirements for the artefacts.

Considering the grid as a graph gives a richer insight into the schema. According to

[157] Greenfield, J. Short, K. 2004. *Software Factories* p. 166
[158] Greenfield, J. Short, K. 2004. *Software Factories* p. 165

Greenfield[159], a factory schema *is actually a directed graph whose nodes are viewpoints and whose edges are computable relationships between viewpoints called mappings*. Treating the schema as a graph allows one to understand the relationships between nodes (i.e. viewpoints, or cells in the grid) in greater depth, since one can construct relationships between nodes that are non-adjacent in the grid. The software factory as graph *provides a multi-dimensional separation of concerns based on various aspects of the artefacts being organized, such as their level of abstraction, position within architecture, functionality or operational qualities*[160]. The grid is really no more than a projection of the graph onto a two-dimensional plane.

Greenfield likens the software factory schema to a recipe that *defines the ingredients, tools, and preparation process for a family of software products*.

4.5.1.2 Software Factory Templates

Since the software factory schema provides the recipe, one needs to gather the ingredients to produce what the recipe describes. The software factory template is, metaphorically, equivalent to the actual ingredients described in the recipe.

The template implements the software factory schema in order to build a member of the product family for which the schema has been designed. The implementation process involves the creation of domain specific languages, patterns, frameworks and tools. It includes making the set of artefacts available to developers[161].

Once the software factory template has been created based on the schema, product development can commence.

4.5.1.3 Software Factories – Critical Innovations

Software factories depend on four critical innovations to achieve the degree of automation that is being hoped will be realised by this approach to software development. These innovations are:

- Systematic reuse: Greenfield contends[162] that it is much simpler to create reusable software in specific contexts (i.e. for reuse in a specific domain or industry) than it is to do so in an undefined context. The issue of the difficulty of systematic reuse

[159] Greenfield, J. Short, K. 2004. *Software Factories* p. 166
[160] Greenfield, J. Short, K. 2004. *Software Factories* p. 168
[161] Greenfield, J. Short, K. 2004. *Software Factories* p. 173
[162] Greenfield, J. Short, K. 2004. *Software Factories* p. 125

can be addressed by defining a family of software products, *whose members vary, while sharing many common features*. Software product lines are an innovation that has been specifically developed to support contextual systematic reuse[163].

- Development by assembly: this refers to the capability of creating new products by assembling pre-existing components. Greenfield states that the ability to do development by assembly has been a feature of many industries, that allowed *craftsmanship to meet the demand of an increasing industrialised society*[164], and that software development is no different. A number of technical advances support development by assembly in the modern software development context, including platform independent protocols, and architecture driven development.
- Model driven development refers to a development process that uses models to automate the development processes. The development of tools and languages that allow developers to express abstractions, and thus create models, makes it easier to create the models required for the automated development processes that model driven development can offer[165].
- Process framework: Greenfield defines a process framework as *a structure that organizes the micro processes used to build the artefacts that comprise the product family members*. This organising framework must have the ability to preserve agility, while managing to scale to accommodate very large scale projects.

4.5.1.4 Innovation and Application

Software factories attempt to bring increased productivity, reliability and reproducible results to software development by effectively industrialising the process of software development. The key innovation is to produce schemas and templates that can be used to create a software product line for a single family of software products. By restricting the scope of the software factory, one can make safe assumptions about what the factory needs to produce. The problem that must be solved (i.e. developing a factory to produce software with the required features and functions) becomes smaller and more

[163] The following definition from Greenfield, page 347, is used to define a software product line: *From a marketing perspective, a product line is a set of products with a common, managed set of features for some target market segments. A software product line adds the concept that the products are software products built from a common set of production assets. The production assets capture knowledge about how to produce the family members, and make it available for reuse to produce the family members.*

[164] Greenfield, J. Short, K. 2004. *Software Factories* p. 110

[165] Greenfield, J. Short, K. 2004. *Software Factories* p. 104

manageable. And because the domain is restricted, it is possible automatically perform certain optimisations automatically, rather than manually – which is an error-prone procedure[166].

Software factories can incorporate Agile development methods, although the nature of software factories require an adaptation of the way that Agile is practised. Software factories do provide a more robust environment in which to practise Agile methods, because the effect of changes in an entire product family can be assessed more quickly, than in a standalone product development scenario[167].

The innovation represented by systematic reuse, in particular, gives rise to economies of scope as expressed in software product lines. Software product lines identify common problem features for product families, they identify and implement common solution features for the product family, and they help to define a mapping between variable problem and solution features for a product family[168]. This allows software product lines to provide at least a modest degree of systematic reuse (and sometimes a very high degree of reuse) thereby promoting economies of scope.

Greenfield's vision for software factories, as an enabler of industrialisation in the software industry, has a number of implications for the industry, and will change the way the software industry works if the approach is widely adopted.[169] It is proposed that development by assembly will become widespread – developers will create only a small part of each application, and would rather reuse existing components. Software supply chains for components will become established. Domain specific assets will come into being. Organisational changes in the industry will be marked by formalised professional qualifications for software professionals. Mass customisation of software may eventually become commonplace. These changes are no less than one would expect as symptomatic of increasing industrialisation of a particular industry.

The software factory approach is based on the convergence of key ideas in the areas of systematic reuse, development by assembly, model driven development and process frameworks[170]. Although many of the ideas are not new, the integrated approach itself is

[166] Greenfield, J. Short, K. 2004. *Software Factories* p. 565
[167] Greenfield, J. Short, K. 2004. *Software Factories* p. 584
[168] Greenfield, J. Short, K. 2004. *Software Factories* p. 348
[169] Greenfield, J. Short, K. 2004. *Software Factories* p. 185 - 189
[170] Greenfield, J. Short, K. 2004. *Software Factories* p. 190

not yet a mature technology. Greenfield[171] enumerates language technology, tool extensibility, pattern composition, deferred encapsulation and development standards for domain specific languages, patterns, frameworks and tools as the most immature aspects of this approach. This list comprises many of the core technologies that are crucial to the technical development of software factories.

Greenfield believes that software factories will initially be adopted to develop business applications – specifically enterprise applications. The reason for this is that these types of applications lend themselves to being analysed for recurring patterns and themes. He suggests that the gradual adoption and successful implementation of products using software factories will lead to larger scale adoption. It is deduced from this statement that the maturation of the software factory approach will be driven by economic imperatives – Greenfield likens the drive towards this transition to the same drive that moved the software development community from developing with very low level languages, to development with higher level, more abstract languages[172].

The preceding discussion is an extremely brief overview of a complex subject, but it is intended only to show that valid technical approaches exist for advancing the industrialisation of the industry.

The next section discusses the concept of cloud computing, which represents an approach to presenting and consuming software and other information technology resources.

4.5.2 Cloud Computing

Cloud computing is a term that is coming into use at the time of writing, to describe a deliver model for a number of information technology related services. Gartner[173]

[171] Greenfield, J. Short, K. 2004. *Software Factories* p. 589

[172] Software development in the earliest days of software application development was often done using so called assembly language. This is a form of coding that requires very little translation by a compiler, in order for it to be executable on a computer – i.e. it is very close to *machine language*, which is the native code that can be directly executed by a computer. Although assembly language led to programs that were optimised for performance, they are difficult to write, and time consuming to produce. The development of higher level languages such as C, C++ and later Java and others, led to much greater productivity, though at the cost of less optimal machine code. Economical imperatives nevertheless drove the transition from assembly language to higher level languages - the higher productivity of scarce resources, and the competitive advantage to be gained from being able to develop software faster, eventually motivated compiler technology to be improved to the extent that code developed in assembly language, was no more effective than code developed using higher level languages. (Greenfield, J. Short, K. 2004. *Software Factories* p. 588)

[173] Plummer, D.C., Bittman, T., J., et al, 2008. *Cloud Computing* p. 3

defines *Cloud Computing* as *a style of computing where massively scalable IT-enabled capabilities are delivered 'as a service' to external customers using Internet technologies.*

This definition is complemented with that of Wikipedia, which states that cloud computing is *a style of computing where IT-related capabilities are provided as a service, allowing users to access technology-enabled services* in the cloud *without knowledge of, expertise with, or control over the technology infrastructure that supports them*. Combining these two definitions, one finds that cloud computing is:

> *Combining the definitions, we find that cloud computing is a style of computing where massively scalable, IT-enabled capabilities are delivered as a service to customers and where the customers require no knowledge of the infrastructure that supports the services.*

This definition incorporates a number of key concepts. Those concepts are:

- **Cloud computing delivers a service**. Gartner states[174] that cloud service must offer a service in order to qualify as such. *The cloud exists as a function of a vendor's ability to deliver services to consumers. This means that the differenc between entities such as the Web or the Internet and the cloud is a matter of intent, rather than kind*[175].
- **Cloud computing is scalable**. Cloud computing is delivered to any number of customers (although the universe of customer may be limited by the service provider). It is therefore incumbent on the service provider to ensure that the service can handle the maximum number of users that could potentially require the services of the cloud. This attribute is referred to as *scalability*[176].
- **Cloud computing serves *customers***. Gartner states that the delivery of services to customers causes an implicit (possibly supplemented with an explicit) service level agreement to come into existence between the service provider and the consumer[177].
- **Cloud computing hides technological complexity from users**. The definition states that users need have no knowledge of the infrastructure that supports the

[174] Plummer, D. C., 2008. *How to identify Cloud Computing*. p. 3
[175] Plummer, D. C., 2008. *How to identify Cloud Computing*. p. 3
[176] Plummer, D.C., Bittman, T., J., et al, 2008. *Cloud Computing* p. 4
[177] Plummer, D. C., 2008. *How to identify Cloud Computing*. p. 3

services. This directly implies that the cloud offering hides technological complexity from users, insofar users may consume the offering without reference to the implementation of the cloud.

It is deduced that a cloud is a service offering that allows customer to utilise technology of which they have no knowledge, nor control over. The motivation for making use of cloud offerings are based on a value-proposition tied to the achievement of economies of scale by cloud service vendors[178]. The cloud vendor achieves economies of scale due to the typically large size of the vendor (and the concomitant ability to negotiate better prices from its own vendors), as well as the ability of the vendor to standardise on the components of its own infrastructure[179].The economies of scale that the vendor achieves, allows it to offer services to customer at a lower unit price than what the customer would be able to achieve when maintaining the service infrastructure on its own behalf[180]. The exact manner in which the services offered by the cloud will be monetized by vendors are not entirely clear at the time of writing, but research is being undertaken to determine the best approaches to this problem[181].

It should be noted that the nature of the services offered by the cloud is not stipulated in the definition. The term cloud computing embraces a large number of different services – this reflects the manner in which cloud computing relies on a diverse set of technologies as enabler[182]. Currently a variety of services are considered to be cloud services. These services include the following[183]:

- **Grid computing** – grid computing offers resource to execute massively parallel computational tasks. Grids are widely used in academic and research establishments.
- **Utility computing** – Utility or metered services are very similar to the utility services provided to households, in the sense that the consumer pays for the service on the basis of the time or quantity that is consumed. Amazon web services[184] (commonly known as AWS) are an example of utility services.
- **Software as a Service (Saas)** – SaaS refers to the idea that software is installed at a

[178] Scholler, D. Scott, D. *Economies of Scale Are the Key to Cloud Computing Benefits* p. 1
[179] Scholler, D. Scott, D. *Economies of Scale Are the Key to Cloud Computing Benefits* p. 2
[180] Scholler, D. Scott, D. *Economies of Scale Are the Kry to Cloud Computing Benefits* p. 2
[181] Broberg, J., Venugopa, S., Buyya, R. 2007. Market-oriented grids and utility computing.
[182] Scholler, D. Scott, D. *Economies of Scale Are the Key to Cloud Computing Benefits* p. 3
[183] Plummer, D.C., Bittman, T., J., et al, 2008. *Cloud Computing* p. 4-5
[184] http://aws.amazon.com

central location, and then offered for use by many consumers. The software is being consumed as a service, and the user is billed for the software usage based on time, level of storage utilisation or some other parameter. The offering of *Salesforce.com*[185] stands as an example of this kind of cloud computing.

Of particular relevance to this research is the impact that cloud computing may have on the development of software. Software developers have to take cognisance of a number of factors that influence the way they have to build the software. These factors include scalability, security and privacy, and reliability[186].

It is expected that a single vendor will need to standardise on the interfaces it provides to the consumers of its services – both on a software and a user interface level. We deduce that this will enforce a certain discipline on software developers in terms of the standards they have to work towards. The problem of scaling is a challenge that will have to be overcome for every service that is offered in the cloud – the ubiquity of this problem may additionally lead to the formulation of standard ways of addressing it.

Although cloud computing is made viable from a business perspective by the financial advantages that accrue to both the vendor and the consumer via the achievement of economies of scale, it is proposed that the economies of scope in the software development process will enable vendors to offer even lower unit prices for their consumers by lowering the cost of creating software.

For these reasons cloud computing may add to the incentive to industrialise the software development process.

4.6 Conclusion

In this chapter it was shown that industrialisation can be recognised by noting changes in the production process that lead to greater productivity or manufacturing efficiencies, due to the appearance of economies of scale or scope.

Following Greenfield's arguments, it was shown that the industrialisation of the software industry will be realised due to the appearance of economies of scope, rather than that of scale.

There exists a general consensus in the software industry that a number of problems

[185] http://www.salesforce.com
[186] Hayes, B. 2008. *Commun. ACM* p. 11

exist around the processes and methods used to develop software, as manifested in problems with quality and cost overruns. The industry has been aware of these problems since at least 1968, yet has not been able to come to an agreement on how to address the issues.

A number of people advocate a craftsmanship based approach to software development as the only feasible means of addressing the problems inherent in the industry. They reason that the software industry is fundamentally different from other industries, and can never benefit from industrialisation.

In opposition to this school of thought is a grouping of people who belief that the software industry is different from most other industries, but that these differences do not preclude the industry from being industrialised. These authors argue that industrialisation is inevitable because of demand side economies – demand will simply outstrip the ability of any craftsmanship based industry to satisfy it. Given the ubiquity of software in the world, and the role software place in infrastructure, it follows that economic imperatives will force the industrialisation.

A common thread in the pro-industrialisation argument is the acknowledgement that domain specific knowledge plays a very important part in software development – this fact will play a crucial part in the industrialisation process. Domain specific knowledge is codified as domain specific languages (DSL's) in the context of software development. Software factories make wide use of DSL's in order to define software factory templates. Software factories provide a practical set of technical innovations that shows one way in which the software industry may be industrialised.

Software factories address the production side of software development. The delivery, or consumption side, is addressed by an approach to the delivery of information technology services known as *cloud computing*. Cloud computing describes a service-oriented approach to delivering information technology services (including software based services) to consumers. The cloud computing approach is likely to encourage the industrialisation of the software production process, due to the nature of the requirements inherent in providing the service. Cloud computing will not determine the form taken by the industrialisation of software development, but software factories may well proof themselves to be an appropriate approach that will address the requirements of the cloud computing paradigm.

The previous two chapters have provided an overview of the software development process, and have illustrated arguments both for and against the industrialisation of the software industry. There exists general agreement that the industry is not industrialised, yet there exists feasible arguments in favour of the possibility of the industrialisation of the industry. Moreover, a possible path towards industrialisation is represented by software factories. Pressure towards industrialisation is represented by the paradigm of cloud computing.

The preceding chapters have demonstrated that the industrialisation of the software industry is at least feasible, although it has not happened yet.

The following chapter outlines the theoretical foundation we will use to support the hypothesis stated at the beginning of this work – namely that if the software industry is moving towards industrialisation then knowledge assets in the format of universal production templates will come into being.

Chapter 5
Theoretical Foundations

This chapter presents a knowledge management perspective on the process of the industrialisation of the software development industry. Previous chapters have shed light on the software development process, and have examined the question of whether the software industry is amenable to an industrialisation process similar to that which other crafts-based industries have undergone. Using theoretical work presented by Ikujiro Nonaka and Max Boisot, this chapter proceeds to examine the process of software industrialisation within a knowledge management theoretical framework.

The chapter first provides and overview of the work of Nonaka and Boisot, tying the theory to the problems that are manifesting themselves in the software development industry.

The concept of knowledge assets as presented by both Nonaka and Boisot are examined in detail, in order to lay the foundation for examining the hypothesis that the possible industrialisation of the software industry will lead to the creation of knowledge assets in the format of universal production templates.

The chapter concludes by presenting an integrated view of the two theories of organisational learning, showing that Boisot's theory can provide an explanatory framework for Nonaka's observation-based model of organisational learning.

5.1 Nonaka

Ikujiro Nonaka's seminal work *The Knowledge Creating Company*[187], has led to a new view of organisational knowledge creation since its first publication. According to Nonaka:

> Organizational knowledge creation theory *attempts to explain the process of making available and amplifying knowledge created by individuals as well as crystallizing and connecting it to an organization's knowledge system*[188]

Nonaka proposed a model of organisational knowledge creation that stands in contrast

[187] Nonaka, I. 1991. *Harvard Business Review*
[188] Erden, Z. von Krogh, G., Nonaka, I. 2008. *Journal of Strategic Information Systems* p. 4

to the (then) accepted Western notions of how knowledge is created within organisation. The Western concept of knowledge creation in the organisation sees the organisation as an information processing machine. The output of this machine is a quantifiable set of data - *codified procedures, universal principles*[189] . Nonaka described a process that is closely aligned to Eastern philosophical principles. The cornerstone of Nonaka's epistemology is the *distinction between tacit and explicit knowledge*[190] . According to Nonaka, knowledge creation is enabled by the conversion of tacit knowledge into explicit knowledge in a spiral movement across *ontological levels*. The ontological levels in the theory proceed from the lowest level, which is that of the individual, to the highest level, which is that of the organisation. Nonaka states that, strictly speaking, knowledge creation only takes place in the minds of individuals – he regards organisational knowledge creation as a process that *amplifies the knowledge created by individuals and crystalizes it as a part of the knowledge network of the organisation. This process takes place within an expanding community of interaction, which crosses intra- and interorganizational levels and boundaries.*[191]

The pattern of knowledge conversion described by the spiral describes a process Nonaka calls the SECI process. SECI is an acronym for socialization, externalisation, combination, and internalisation. These are the four modes of knowledge creation[192]:

- **Socialization** represents the sharing of tacit knowledge, typically in an informal or unstructured manner, i.e. by sharing experiences, or by conversation. Socialization therefore is a process of conferring tacit knowledge amongst individuals.
- **Externalisation** - knowledge is made more explicit by the use of metaphors, concepts, hypotheses or models. This mode converts tacit knowledge to explicit knowledge. Nonaka considers this mode to be the key to knowledge creation, and suggest that sequentially using metaphor, analogy and model provides the most effective and efficient means of converting knowledge from tacit to explicit.
- **Combination -** a *process of systemizing concepts into a knowledge system* - it involves combining different, existing bodies of explicit knowledge. This mode of knowledge conversion converts from explicit to explicit knowledge – new

[189] Nonaka, I. 1991. *Harvard Business Review* p. 164
[190] Nonaka,I., Takeuchi, H. 1995. *Knowledge Management Classic and Contemporary Works* p. 139
[191] Nonaka,I., Takeuchi, H. 1995. *Knowledge Management Classic and Contemporary Works* p. 142
[192] Nonaka,I., Takeuchi, H. 1995. *Knowledge Management Classic and Contemporary Works* p. 146-156

knowledge may be created from the synthesis of existing bodies of knowledge.

- **Internalisation** - represents a conversion from explicit knowledge back to tacit knowledge – it is effectively *learning by doing*. Once explicit knowledge has been internalised, the know-how represented by this knowledge becomes a valuable asset in itself.

The current state of software development is largely predicated on a crafts-based approach, meaning that the knowledge that is required to produce the product resides in the individual team members, and has not been expressed in a manner that easily allows the dissemination of the knowledge This point was discussed in depth chapter four, where it was shown that the software industry has not been industrialised, as shown by the preponderance of a crafts-based approaches to software development.

It may be deduced that the software development industry finds itself in the socialization part of the SECI spiral, meaning that the knowledge held by the workers in the industry, is held at a low ontological level – i.e. that of the individual. According to Nonaka, the next step in the process would be that of externalisation. Given the discussion in chapter three of the characteristics of scrum development (including organisational transfer of learning[193]) it appears that in certain approaches to software development, movement along the knowledge spiral is manifesting itself – externalisation is manifesting as organisational learning.

[193] Nonaka, I. Takeuchi, H. 1986. *Harvard Business Review*. p. 144

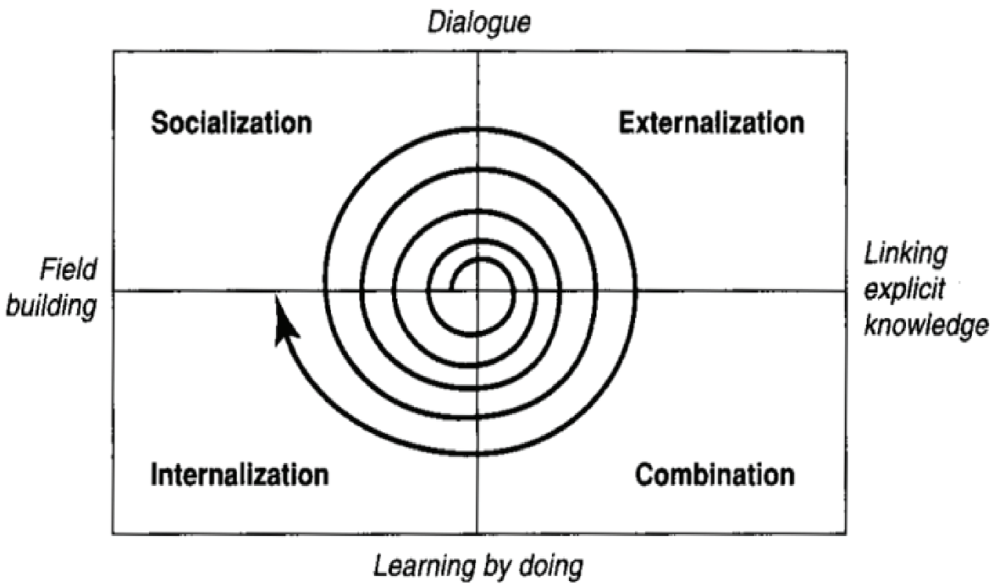


Figure 5.1 The SECI spiral

Source: Nonaka, 1995

The diagram above (figure 5.1) illustrates the SECI spiral, showing the ontological and epistemological dimensions in which the spiral of knowledge creation occurs.

The process of organizational *amplification* of knowledge during the process of knowledge conversion between successive ontological levels is shown in figure 5.2, below.

The spiral of knowledge creation shown in figure 5.2 illustrates the way in which organisational knowledge creation *is a spiral process, starting at the individual level and moving up thought expanding communities of interaction, that crosses sectional, departmental, divisional, and organisational boundaries*[194].

[194] Nonaka,I., Takeuchi, H. 1995. *Knowledge Management Classic and Contemporary Works* p. 143

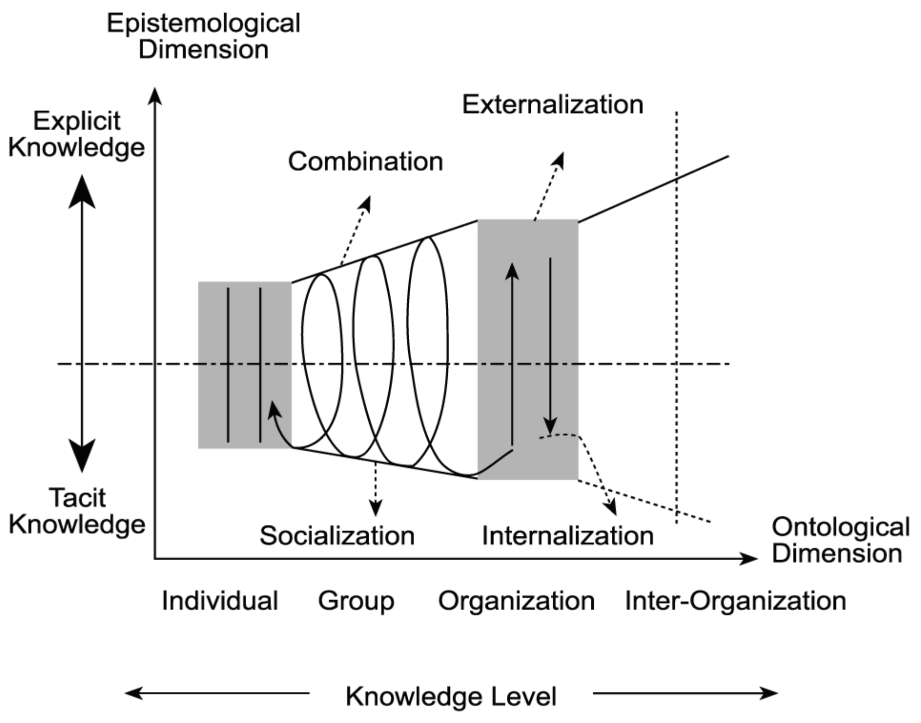


Figure 5.2: Spiral of organisational knowledge creation

Source: Nonaka 1995

Nonaka identifies five enabling conditions for organisational knowledge creation. These enabling conditions both drive the process of knowledge creation, and are prerequisites to the process. The conditions are[196]:

- **Intention –** organisational intention, the *organisation's aspiration to its goals*[197] (Nonaka 1991, p 159) drives the knowledge spiral in the first instance.
- **Autonomy**, i.e. the freedom for individual member of the organisation to act autonomously as far as possible, drives the process by allowing individuals to introduce and discover opportunities. The scrum process, discussed in chapter two, is an example of a software development approach that allows a large degree of freedom for the individual team members[198].
- **Fluctuation and creative chaos** is essential for stimulating the interaction between the organisation and the external environment.
- **Redundancy –** the existence of information beyond what is immediately needed by

[196] Nonaka,I., Takeuchi, H. 1995. *Knowledge Management Classic and Contemporary Works* p. 159-171
[197] Nonaka,I., Takeuchi, H. 1995. *Knowledge Management Classic and Contemporary Works* p. 159
[198] Nonaka,I., Takeuchi, H. 1995. *Knowledge Management Classic and Contemporary Works* p. 164-165

individuals – promotes the sharing of tacit knowledge.

- **Requisite variety** is the final enabler – *an organisation's internal diversity must match the variety and complexity of the environment in order to deal with challenges posed by the environment*[199].

The preceding discussion outlined Nonaka's original theory of organisational knowledge creation. The theory has direct bearing on the way in which organisational learning may contribute to the industrialisation of the software industry, since it shows a route whereby tacit knowledge (craftsmanship) may be transformed to explicit knowledge.

Nonaka expanded the theory of organisational knowledge creation beyond the first formulation outlined above. In particular, he introduced the concept of *ba* and *ART systems* in a 1998 paper entitled *The 'ART' of Knowledge: Systems to Capitalize on Market Knowledge*[200].

Ba is a Japanese philosophical concept, which can be thought of as *a shared mental space for emerging relationships*[201] This space need not be physical, but can be a virtual space, or a mental spaced created by the sharing of ideas and experience. It provides a platform for advancing collective and individual knowledge, and can be thought of as the environmental settings.

ART systems, or action-reflex-trigger systems, are systems that *trigger the dynamic process of accumulation, creation, exploitation and dissemination of knowledge*[202]. The ba in which the ART system functions, determines the type of knowledge creation that would take place[203], i.e. that stage in the SECI conversion process. ART systems attempt to routinize knowledge conversions[204].

As such these systems can be used to encourage the conversion process, facilitating knowledge creation in the organisation[205]. *ART systems are the conceptual foundation for the routines that support organizational knowledge creation*[206].

Routinising the conversion process may contribute to the industrialisation of an industry

---

[199] Nonaka,I., Takeuchi, H. 1995. *Knowledge Management Classic and Contemporary Works* p. 170
[200] Nonaka, I. Reinmoeller, P., Senoo, D.1998. *European Management Journal 16(6)*
[201] Nonaka, I. Reinmoeller, P., Senoo, D.1998. *European Management Journal 16(6)* p. 675
[202] Nonaka, I. Reinmoeller, P., Senoo, D.1998. *European Management Journal 16(6)* p. 673
[203] Nonaka, I. Reinmoeller, P., Senoo, D.1998. *European Management Journal 16(6)* p. 680
[204] Nonaka, I. Reinmoeller, P., Senoo, D.1998. *European Management Journal 16(6)* p. 677
[205] Nonaka, I. Reinmoeller, P., Senoo, D.1998. *European Management Journal 16(6)* p. 681
[206] Nonaka, I. Reinmoeller, P., Senoo, D.1998. *European Management Journal 16(6)* p. 681

– by being able to routinely convert tacit knowledge to explicit knowledge, the knowledge becomes widely disseminated, and move out of the realm of craftsmanship.

In 2000, Nonaka proposed a unified theory of knowledge creation incorporating the concepts of the SECI knowledge creation spiral, *ba*, and knowledge assets. In this contexts knowledge assets are defined as *the inputs, outputs and moderators of the knowledge-creating process*[207]. This extension of the knowledge creation theory describes how an organisation creates new knowledge through the SECI process, and by using existing knowledge assets. The knowledge creation process takes place in ba. New knowledge created in this fashion, becomes the basis for further knowledge creation spirals.

Nonaka identifies four categories of knowledge assets, as described in the table below[208]:

Experiential Knowledge Assets Tacit knowledge shared through common experiences - Individual skill and know how - Trust and love - Energy, passion and tension	**Conceptual Knowledge Assets** Explicit knowledge articulated through images, symbols and language - Product concepts - Design - Brand equity
Routine Knowledge Assets Tacit knowledge routinised and embedded in actions and practices. - Operational routines - Organisational culture	**Systemic Knowledge Assets** Systemised and packaged explicit knowledge. - Documents, specifications, manuals - Databases - Patents and licenses

Knowledge assets from all the categories defined above are both inputs to the organisational knowledge creation process, as well as an output of this process – in this way these knowledge assets are not static, but are continuously evolving. Nonaka's definition of knowledge assets will be compared to that of Boisot in this chapter, in order to clearly define the nature of, and role knowledge assets in the knowledge creation process.

A paper published in 2003[209] proposes a further elaboration of the knowledge creation theory by means of viewing the organisation as a dialectic being. This view of the

[207] Nonaka, I., Toyama, R., Konno, N. 2000. *Long Range Planning 33* p. 20
[208] Nonaka, I., Toyama, R., Konno, N. 2000. *Long Range Planning 33* p. 20
[209] Nonaka, I., Toyama, R. 2003. *Knowledge Management Research and Practice 1*

organisation necessitates considering the process of knowledge creation activities within the organisation, rather than just the outcome of these activities. Quoting from the paper[210]:

> *The main line of thought in this paper is that a firm is a dialectic being that synthesizes various contradictions through SECI and ba, and strategy and organization should be re-examined from such an integrated viewpoint instead of logical analysis of structure or action. An organization is not a collection of small tasks to carry out a given task, but an organic configuration of ba to create knowledge.*

The view of the organisation as a dialectic being, and Nonaka's attempt to incorporate Giddens' Structuration Theory into a knowledge creation theory of the organisation has attracted some criticism. Zhu[211] argues that Nonaka's incorporation of Giddens is too uncritical, and does not address concerns Giddens himself have raised regarding aspects of Structuration Theory.

The most recent extension (at the time of writing) to Nonaka's organisational knowledge creation theory has been proposed by Erden, Nonaka and von Krogh , in a 2008 paper addressing the issue of the quality of tacit knowledge within a group[212]. The authors elaborate on the concept of group tacit knowledge, and show that it is possible to attach meaning to the quality of group tacit knowledge – a measure of quality can be derived by examining the way in which the group as a whole acts – at the lowest quality level the group is merely a collection of people, while at the highest quality level of group tacit knowledge, the group show evidence of *collective improvisation*[213]- the ability to deal as a group (without the requirement of explicit leadership) with all but the most unfamiliar situations. The concept of group tacit knowledge has not been explored much beyond the conceptualisation[214], but the concept provides a viable basis for exploring the creation of explicit knowledge during the software development process, particularly when considering Agile development processes such as scrum. Group tacit knowledge may be considered to be at a higher level of diffusion than individual tacit knowledge – this will be examined further in the context of Boisot's notions of diffusion

[210] Nonaka, I., Toyama, R. 2003. *Knowledge Management Research and Practice 1* p. 9
[211] Zhu, Z. 2006. *Knowledge Management Research and Practice 4*
[212] Erden, Z. von Krogh, G., Nonaka, I. 2008. *Journal of Strategic Information Systems*
[213] Erden, Z. von Krogh, G., Nonaka, I. 2008. *Journal of Strategic Information Systems* p. 13
[214] Erden, Z. von Krogh, G., Nonaka, I. 2008. *Journal of Strategic Information Systems* p. 5

in the knowledge creation process.

Nonaka's theory of organisational knowledge creation is based on observation of how the organisation functions[215] In contrast to the approach that Nonaka takes, is Max Boisot's contributions towards the formulation of a knowledge-based theory of the organisation. We discuss the formulation of Boisot's theory in the following section.

5.2 Boisot

Max Boisot proposed a theoretical framework[216] which can from the basis for a knowledge-based theory of the organisation. The theoretical framework that Boisot developed is known as the I-Space. The I-Space uses the nature of information and knowledge flows in any system as a point of departure[217] for modelling the knowledge creation process. This is a more abstract approach than the approach taken by authors such as Nonaka, who approach the model from a standpoint of observing what the members of an organisation actually do[218]. Boisot’s framework is scalable in the sense that it contains no inherent limitations as to the level at which it can be applied – it is equally suited to application at the level of the organisation, as to application at the level of entire societies[219]. The theoretical foundations on which the I-Space conceptual framework is based, hinge on the distinction amongst data, information and knowledge. The I-Space framework allows us to analyse the flow properties of information *within different agent groupings, as a function of its degree of codification and abstraction*[220]. We proceed by presenting Boisot's analysis of the relationships amongst the concepts of data, information and knowledge, and we then provide an overview of the I-Space conceptual framework. The propositions of Boisot's framework are related to the known problems with software development throughout the discussion, and implications relevant to the industrialisation of the software development process are pointed out.

5.2.1 Data, Knowledge and Information

Boisot draws a very strong distinction between data and information, and information and knowledge. According to him physicists tend to conflate the concept of data and

[215] Boisot, M. H. 2007. *Explorations in Information Space* p.6
[216] Boisot, M.H. 1995. *Information Space*
[217] Boisot, M. H. 2007. *Explorations in Information Space* p. 7
[218] Boisot, M. H. 2007. *Explorations in Information Space* p. 7
[219] Boisot, M. H. 2007. *Explorations in Information Space* p. 8
[220] Boisot, M., H. 1998. *Knowledge Assets* p. 41

information, and similarly, social scientists conflate information and knowledge[221]. This conflation matters, because if the three concepts are treated as separate from each other, it becomes possible to exploit the differences amongst the concepts in the context of economic theorizing, opening the way towards treating knowledge as a resource in an economic model.

The following definitions related to data, information and knowledge are taken from Boisot[222]:

- **Data** can be viewed and treated as having its origin in the physical world – data originates from discernible differences in physical states of the world, states which are describable in terms of space, time and energy.

- **Information** is extracted from data, by an information processing agent[223]. Because data originates from state changes in the real world, it is possible that not all data will register as stimuli for information processing agents (the limits of an agent’s perception being related to the senses available to the agent – human beings, for example, cannot perceive infra-red colours – yet these colours are data according to our definition). Those stimuli that do register with an agent, is subject to neural processing before it is recognised as data by the agent – this kind of processing requires energy expenditure. Information, then, constitutes those significant regularities *residing in the data that agents attempt to extract from it*. The term *significant regularity* is subjective – its meaning is likely to differ amongst agents. The definition of information Boisot proposes, therefore implies that information effectively sets up a relation between incoming data and a given agent. Boisot notes that only once convention has established what is meant by *significant regularity*, can information appear to be objective[224].

- **Knowledge** is held to be *a set of expectations held by agents and modified by the arrival of information*. *Expectations* in this context are the agent's prior

[221] Boisot, M. H. 2007. *Explorations in Information Space* p. 16-17
[222] Boisot, M. H. 2007. *Explorations in Information Space* p. 19-20
[223] In the context of the I-Space, an agent is any system that receives, processes and transmits data with sufficient intelligence to allow learning to take place. *Whatever exhibits intelligent agency is an agent for our purposes*. Boisot's definition of an agent is precisely what makes the I-Space framework very scalable – the population of agents can vary form a collection of neurons, to a population of human beings of any size and nature.
[224] Boisot, M. H. 2007. *Explorations in Information Space* p. 19

learning.

In summary: *information is an extraction from data that, by modifying the relevant probability distributions, has a capacity to perform useful work on an agent's knowledge base.*[225]

Boisot's modelling of the relationship between data and information, and information and knowledge is important in order to precisely define the nature of knowledge assets, as well as the knowledge creation process that leads to the creation of knowledge assets. Since knowledge assets are central to the hypothesis of this research, it is meet to define the concept from basic principles.

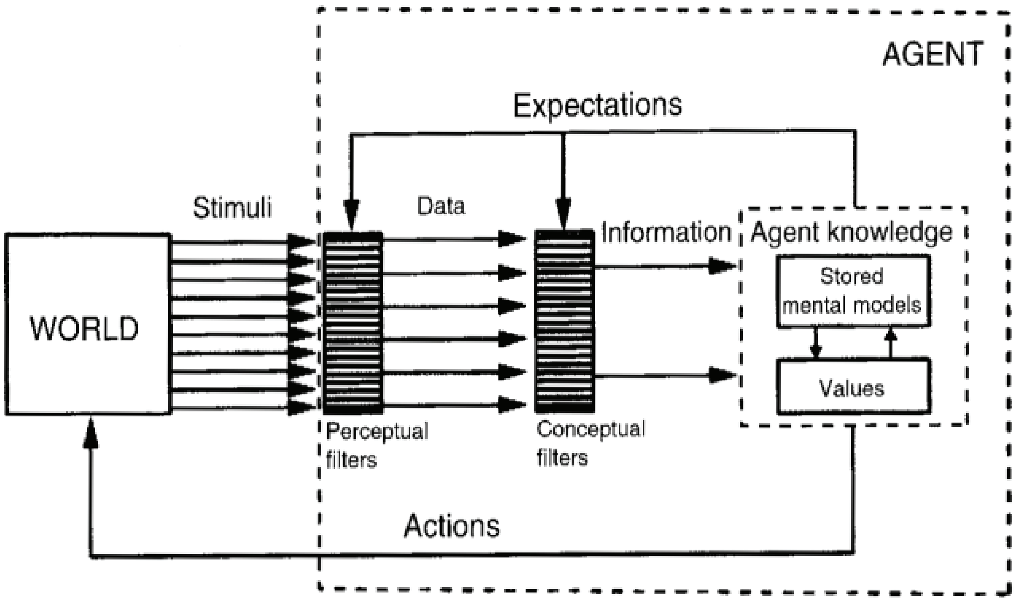


Figure 5.3: The Agent-In-The-World

Source: Boisot (2007)

Figure 5.3, above, shows the essential relationships amongst the concepts *data*, *information* and *knowledge*. According to Boisot agents operate two kinds of filters. The first set of filters is the set of perceptual filters – these filters only allow some kinds of stimuli to be recognised as data by the agent – such filters may represent physical limitations to perceptual awareness. The second, conceptual, set of filters aer those that extract information-bearing data from the stimuli (data) that has passed through the perceptual filters. The effect of both sets of filters are adjusted, or tuned, by the agent's

[225] Boisot, 2007, page 20

cognitive and affective expectations to act in a selective fashion on stimuli and data[226].

In terms of the commonly occurring failure to distinguish amongst data, information and knowledge, Boisot demonstrates the manner in which physicists tend to conflate data and information by looking at both information theory and information physics. Information physics is a relatively new discipline that *researches the physical laws governing the acquisition, processing, and transmission of information*[227] at a quantum level. He demonstrates the data-information conflation in information physics by expounding on the differences between thermodynamic entropy and information entropy, and showing how the same laws of physics are applicable to both types of entropy, even though information and thermodynamic entropy are fundamentally different from each other. The conclusion Boisot derives from this, is that physicists do not distinguish between data and information[228]. Turning to information theory, Boisot points out that data and information is treated as virtually identical, by assumption. Because information theory explicitly sets out to address the technical problem (the so called level one problem) of communication, it avoids addressing the semantic and effectiveness aspects of communications[229], both of which are associated with knowledge.

The conflation of knowledge and information in the social sciences is demonstrated by the manner in which two disciplines, economics and sociology, treat these two concepts. In neoclassical economics, the assumption is made that no agent is subject to data processing limitations, thereby completely avoiding the responsibility of addressing the nature of information or knowledge. Neo-classical economic theory has not been borne out in practice – human beings do in fact have limitations in terms of data processing. Evolutionary economics[230] holds a more realistic view of the role of knowledge, but

[226] Boisot, M. H. 2007. *Explorations in Information Space* p. 20

[227] Information Physics – University of New Mexico. http://info.phys.unm.edu/

[228] Boisot reaches this conclusion by considering Landauer's principle, which describes the connection between thermodynamic and information entropy. (Boisot, M. H. 2007. *Explorations in Information Space* p. 31-34). The conflation arises because there is no clear recognition of the the fundamental differences between the two types of entropy. Thermodynamic entropy concerns itself with data – regularities or the lack thereof in discernible states of the world, while information entropy refers to *the information that can be extracted from such states by a knowledgeable observer*.

229 The level 2, or semantic problem is the problem of knowing whether the received message was understood. The level 3 problem that of effectiveness, questions whether the message led to the desired behaviour. Levels 2 and 3 are associated with knowledge.

[230] Evolutionary economics is an approach to economic analysis that discards the primary tenets of orthodox economics – i.e. the rational allocation of scarce resources by an agent, with the goal of maximising her welfare. Evolutionary economics treats economics as an evolutionary dynamics, rather than a system mainly concerned with finding equilibrium. Agents in evolutionary economics

nevertheless implicitly assumes a very tight coupling between knowledge and information[231]. Sociologists on the other hand have tended to concentrate on knowledge alone, without reference to either information or data. Although organisational sociology has addressed the process whereby information is interpreted and converted to knowledge via the study of sensemaking[232], it does not address the nature of data.

Boisot demonstrates that the concepts of data, information and knowledge are most often not properly distinguished from one another. Making such a distinction is a pre-requisite for developing an economic model that incorporates knowledge as an endogenous factor, rather than as an exogenous factor in the manner of orthodox economics. Once an economic model has been established, it will become possible to develop a knowledge-based theory of the firm. We proceed in the next section to describe Boisot's derivation of such an economic model. Boisot's economic model of knowledge creation is essential to understanding the effect that industrialisation will have in a given industry, since industrialisation involves the consumption of information resources. This is especially true of an industry that uses mainly information resources – of which the software development industry is a good example.

5.2.2 The Economics of Information

Economics has until very recently concentrated on studying the energy economy, and has only recently taken cognisance of the existence of the information economy[233]. In the analysis of the energy economy the factors of production are taken to be either land or labour (in an agricultural economy), or labour and capital (in an industrial economy). The following discussion focuses on industrial economies, where labour and capital are the main factors of production. In both classical and neoclassical economics the impact of knowledge and information is treated as exogenous to the system, meaning that the impact is an input parameter to the system – the model does not attempt to explain the effect, but simply reflects it. A typical production function shows how the factors of production (capital and labour in an industrial economy) can be substituted for each other, while maintaining a constant level of output.

discover and accumulate higher survival value for a given incurred cost, than could be offered by competing alternatives.

[231] Boisot, M. H. 2007. *Explorations in Information Space* p. 22

[232] Weick, 1995. *Sensemaking in Organizations*

[233] Boisot, M., H. 1998. *Knowledge Assets* p. 24

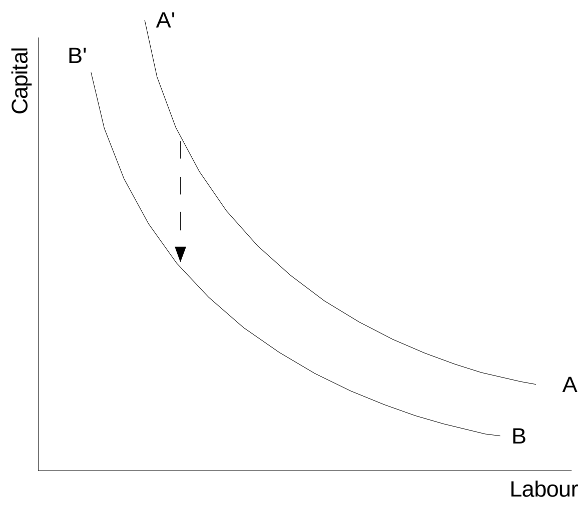


Figure 5.4 Neoclassical production function

Source: Adapted from Boisot (1998)

Figure 5.4, above, shows a production function from neoclassical economics. Each curve (AA' and BB') represents an isoquant. An isoquant shows all combination of capital and labour that will produce the same level of production output. Movements along the isoquant (from A to A' or from B to B' and vice versa) establish the rate of substitution between the factors of production. Technical progress reduces the combined input required to achieve a given level of production – and causes a shift in the isoquant towards the origin – from AA' to BB'. This is an exogenously given discontinuity, not accounted for in the model itself.

Boisot points out that the factors of production (labour and capital) already implicitly incorporate information and knowledge – these factors contain both physical and information attributes. The combinations of capital and labour that are required to produce a given level of output *are a function of the technology, skills and organization available*[234]. Since knowledge, information and data is inextricably bound up with the presentation of the factors of production in the orthodox production function (pointing to a conflation of the concepts, as discussed in the previous section) it is not feasible to expand the production function to explicitly include information as a factor of production – doing so is likely to lead to double counting, and amounts to little more

[234] Boisot, M., H. 1998. *Knowledge Assets* p. 25

than *intellectual legerdemain*[235].

Boisot proposes the introduction of a completely new production function that is capable of reflecting the role of data, information and knowledge as endogenous forces[236]. He argues that such a production function will have to operate at a higher level of abstraction than the orthodox production function. The abstraction can be achieved by grouping all the physical (non-information) factors of production on one axis (these include space, time and energy), and the information and knowledge related functions on the other axis[237]. It is however not appropriate to include information (nor knowledge) as an explicit factor of production – instead data should be included. Based on the distinctions that were drawn earlier between these concepts, it is clear that information is extracted from data, and that information economizes on the use of data if properly utilised. Since both information and knowledge economize on the consumption of data, both lead to a shift in the isoquant of this new production function towards the origin.

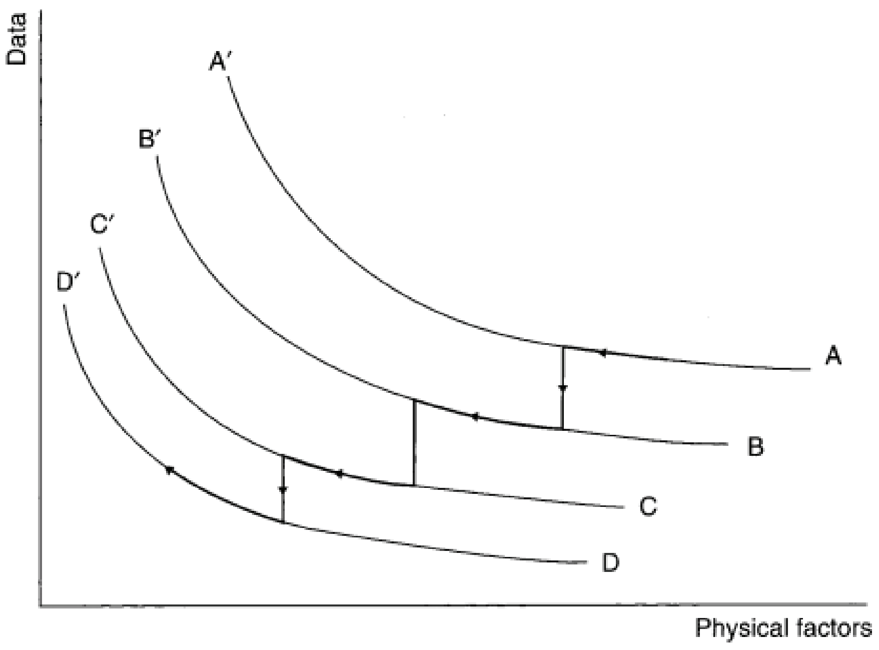


Figure 5.5: Data versus Physical Factors of Production

Source: Boisot, 2007

[235] Boisot, M., H. 1998. *Knowledge Assets* p. 25
[236] Boisot, M., H. 1998. *Knowledge Assets* p. 25
[237] Boisot, M., H. 1998. *Knowledge Assets* p. 26-27

Boisot calls this production function the *evolutionary production function* – it is shown in Figure 5.5. In the evolutionary production function a shift along an isoquant, from right to left, represents a substitution of data for physical factors. This is effectively *learning-by-doing*[238]. Since learning by doing implies memory, a shift from left to right can be interpreted as *forgetting, an erosion of memory, or as the workings of bounded rationality*[239]. Movement to either the left or the right is possible. A movement towards the origin, downwards and across isoquants, represent *the generation of insight, the extraction of information from data to create new, more abstract knowledge concerning the structure of the underlying phenomena*[240]. Movements across isoquants are discontinuous, reflecting the unpredictable nature of creative insights. Discontinuous jumps towards the origin represent the creation of new knowledge, while movement along an isoquant are representative of the application of knowledge[241]. Movement along an isoquant represents technical change, and movement across isoquants represents technical progress. In both the neoclassical and the evolutionary production function, the movement across isoquants, towards the origin, is interpreted in the same way. Boisot notes, however, that the evolutionary production function differs from the neoclassical function in three important ways.

- The evolutionary production function imparts a *preferred direction* to the trade-off between physical and data factors of production, albeit a global, not local bias[242]. The bias is towards substituting data resources for physical resources. Boisot reasons that this is because evolution selects for intelligence, which in turn demonstrates a bias towards data over physical resources[243].
- Unlike the neoclassical production function, the evolutionary production function *can account for technical progress*. The jump between isoquants can be explained in terms of the discontinuous manner in which living organisms learn – the discontinuous phenomenon of insight[244].
- The factors of production in the evolutionary production function exhibit *very different economic properties* compared to those of the neoclassical production

[238] Boisot, M., H. 1998. *Knowledge Assets* p. 30
[239] Boisot, 2007, page 37
[240] Boisot, 2007, page 37
[241] Boisot, M., H. 1998. *Knowledge Assets* p. 34
[242] Boisot, M. H. 2007. *Explorations in Information Space* p. 38
[243] Boisot, M. H. 2007. *Explorations in Information Space* p. 37
[244] Boisot, M. H. 2007. *Explorations in Information Space* p. 38

> function. In the neoclassical production function, all the physical factors of production are naturally subject to scarcity constraints – but this is not the case where data factors are concerned. Data factors are scarce only insofar that living systems have a limited capacity for receiving, storing, processing and transmitting data – they are not scarce by nature[245].

Bearing in mind that knowledge assets are defined as accumulations that yield a stream of useful services while economising on the consumption of physical resources[246], it is apparent that knowledge assets emerge as useful productive structures that are located on an isoquant such as AA' in figure 5.5[247].

The evolutionary production function requires an act of abstraction in order to conceptualise it. This abstraction adds to the generality of the theory, but what is gained in generality is lost in specificity. Boisot cautions that a detailed specification of the physical and information inputs into the evolutionary production function will be required for practical applications[248].

Using the concept of the evolutionary production function, it becomes possible to examine the way in which dominant designs come into being. Dominant designs are of particular interest in the study of industrialisation, as discussed below.

5.2.2.1 Dominant Design

Industrialisation requires that greater productivity and improved manufacturing efficiencies should appear in the manufacturing process, before it is recognised in an industry. Additionally, industrialisation is typified by the emergence of standards – and a dominant design implies standardisation: the majority of industry members will use the same process or methodology if a dominant design has come into being. These topics were discussed in depth in chapter four. *It is proposed that the process by which a dominant design emerges and stabilizes may be interpreted to reflect a process of industrialisation, if the ontological level is set to that of the industry.* This proposition is discussed in the section below.

[245] Boisot, M. H. 2007. *Explorations in Information Space* p. 38
[246] Boisot, M., H. 1998. *Knowledge Assets* p. 13
[247] Boisot, M., H. 1998. *Knowledge Assets* p. 26
[248] Boisot, M., H. 1998. *Knowledge Assets* p. 27

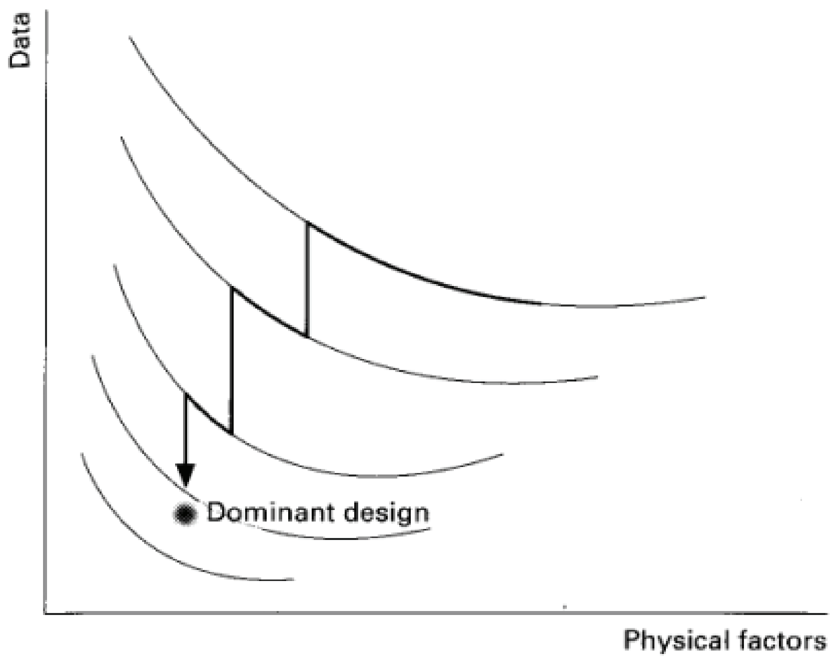


Figure 5.6: Dominant Design

Source: Boisot (1998)

Figure 5.6 shows the way in which a dominant design is established in the evolutionary production function.

The movements along isoquants represent complexity absorption, and those across isoquants represent complexity reduction[249]. The complexity reduction effect is driven by the manner in which standardisation affects product development. As more users use a product, the standard features and performance improvements tend to focus on those features that are valued by the greatest number of customers[250]. *Standardisation leads to performance improvements, which in turn leads to further standardisation.* During this process, *product attributes coevolve, to constrain each other mutually*[251]. The combination of performance improvements and standardisation lead to factor savings because as reliability increases, fewer parts are used, and a growing percentage of production can be automated[252]. The movement along isoquants reflect a reduction in the complexity of physical manufacturing processes, but at the cost of increased data -

[249] Boisot, M., H. 1998. *Knowledge Assets* p. 37
[250] Boisot, M., H. 1998. *Knowledge Assets* p 36
[251] Boisot, M., H. 1998. *Knowledge Assets* p 36
[252] Boisot, M., H. 1998. *Knowledge Assets* p 36

processing complexity. The complexity absorption and reduction processes takes place in parallel, but the absorption process is constrained by the standards of the reduction process. Together these two processes lead to the establishment of a dominant design, at the position indicated in figure 5.6.

Boisot has stated that the appearance of a dominant design goes hand-in-hand with improved manufacturing efficiencies. Productivity is defined as *a performance measure that indicates how effectively an organization converts its resources into its desired products or services*[253] for the purposes of this research. By economizing on data consumption (which complexity reduction brings about) the productivity of the organisation is increased – since fewer resources are used to produce the same product. It appears that the development of a dominant design can be identified as a process of industrialisation using the criteria stated at the start of this section, provided that the ontological level encompasses that of an industry, and not just a single organisation.

This discussion has explained Boisot's proposal for handling data as an economic factor of production. The evolutionary production function does not, however, answer the question of how agents obtain information from data. Boisot proposed a conceptual framework, known as the I-Space, to explain this process. The I-Space is described in the next section of this chapter.

5.2.3 The I-Space

The I-Space is a single conceptual framework, first proposed by Max Boisot in a 1995 work entitled "*Information Space: A Framework for Learning in Organisations, Institutions and Culture*"[254]. The I-Space provides a framework within which to address the interaction between the proceses of codification, abstraction and diffusion of information. Boisot's framework is presented by describing each of the dimensions of the I-Space, and then describing how the three dimensions are combined to form the integrated framework. The I-Space framework will be used to integrate the theories of Nonaka and Boisot, to further explore the process of industrialisation as it pertains to the software development industry, and to provide supporting evidence towards proving the hypothesis of this research.

[253] O'Neil, S.L, Hansen, J.W. 2001. *Encyclopedia of Business and Finance* p.708
[254] Boisot, M.H. 1995. *Information Space*.

5.2.3.1 Codification

In its most general form, codification is the act of creating categories that facilitate the classification of phenomena. These categories make it possible to make *clear and reliable* distinctions between states of the world[255]. Coding is the subsequent act of assigning phenomena to categories, once the categories have been created. The effectiveness of the codification can be measured by the ease with which coding takes place – a more effective codification will lead to much easier coding. Effective codification depends on intellectual and observational skill, as well as on the complexity of the phenomena that are the subject of the codification process[256].

Because codification forces a choice to be made (choosing A is the same as not choosing not-A), codification generates a *cognitive and behavioural commitment*[257]*:* by choosing the categories, the agent becomes committed to act within the constraints imposed by those choices. Furthermore – the choice that codification implies shows that it can be thought of as a process for getting rid of excess data, and consequently a process for economizing on data processing (less data means less processing).

Boisot proposes placing tasks (and the knowledge assets required to execute them) along a codification dimension, that is scaled according to a particular definition of complexity, namely algorithmic information complexity. The definition of algorithmic information complexity is as follows: algorithmic complexity is the number of bits of information needed to carry out a given data-processing task[258]. The uncodified end of the scale represents a non-deterministic, borderline chaotic environment that cannot be codified at all. The opposite side of the scale is an extremely ordered, complete codified environment – effectively reduced to a choice between two states. A movement along the codification dimension from the uncodified to the codified end represents an act of economizing on data processing resources[259]. The second dimension of the I-Space framework is the abstraction dimension– which is discussed below.

5.2.3.2 Abstraction

Whereas codification creates categories according to which phenomena can be coded, abstraction performs the different function of reducing the number of categories that are

[255] Boisot, M. H. 2007. *Explorations in Information Space* p. 117
[256] Boisot, M., H. 1998. *Knowledge Assets* p. 43
[257] Boisot, M., H. 1998. *Knowledge Assets* p. 45
[258] Boisot, M., H. 1998. *Knowledge Assets* p. 46
[259] Boisot, M., H. 1998. *Knowledge Assets* p. 46-47

necessary for the coding to take place for a given task. This represents a further economizing on data processing. In the words of Boisot: *(codification) gives form to phenomena, (abstraction) gives them structure*[260].

The degree of abstraction can be measured by using an additional measure of complexity - *effective complexity*. Effective complexity *is measured by the number of bits of information required to specify whatever regularities characterize the task*[261]. Given this definition, the level of effective complexity will vary with the level of abstraction at which the task is specified.

At one end of the abstraction scale are very concrete experiences. These will be dominated by experiences that produce perceptual and local knowledge. Categories will be rich (implying low codification) and the underlying causal structures will not be easy to discern (implying difficulty in abstraction). At the other end of the scale the knowledge produced will be conceptual and non-local[262].

Codification and abstraction do not operate in isolation of each other. Codification facilitates abstraction, and abstraction stimulates codification[263]. Both provide a means of economizing on data, and both assist in articulating knowledge, and making it more shareable, i.e. increasing the diffusibility of the information.

5.2.3.3 Diffusion

Diffusion is the third dimension of the I-Space. Diffusion refers to the diffusibility, rather than the degree of diffusion of knowledge in a given population. This means that this dimension is not concerned with how many agents have received the information, nor with the rate of uptake, but rather with the possibility of transmitting the information[264].

Scaling of this dimension is according to the proportion of a population that can be reached by information with different degrees of codification and abstraction. The population itself is defined in terms of the relevance of the information to them.

Boisot states that technical considerations may affect the diffusibility and consequently the availability of information in a given population. Higher order social and cultural

[260] Boisot, M., H. 1998. *Knowledge Assets* p. 48
[261] Boisot, M., H. 1998. *Knowledge Assets* p. 50
[262] Boisot, M., H. 1998. *Knowledge Assets* p. 5-51
[263] Boisot, M., H. 1998. *Knowledge Assets* p. 51
[264] Boisot, M., H. 1998. *Knowledge Assets* p. 53

considerations will influence the ability of a population to absorb the information – that is the rate at which the information is taken up and used[265].

5.2.3.4 The Framework

The three basic dimensions of the I-Space, together with the scaling of each dimension have been explained in the previous section. Boisot unifies these three dimensions into the conceptual framework known as the I-Space. The I-Space makes it possible to explore the behaviour of information flows, and by exploring this behaviour, the I-Space framework sheds light on how knowledge is created, and how it diffuses through selected populations.

The key hypothesis of the I-Space is:

> *...codification and abstraction are mutually reinforcing and both, acting together, greatly facilitate the diffusion of information*[266].

The counter proposition is as follows: if an item of information is less codified and abstracted, it will take much longer to reach a given proportion of a selected population. In terms of industrialisation, where knowledge has been disseminated across an entire industry, it follows that the degree of codification and abstraction must either be very high, or the dissemination of information would have taken a very long time.

The I-Space can be visualised as a three-dimensional space, within which knowledge flows can be traced out, as shown in figure 5.7.

[265] Boisot, M., H. 1998. *Knowledge Assets* p. 55
[266] Boisot, M., H. 1998. *Knowledge Assets* p. 55

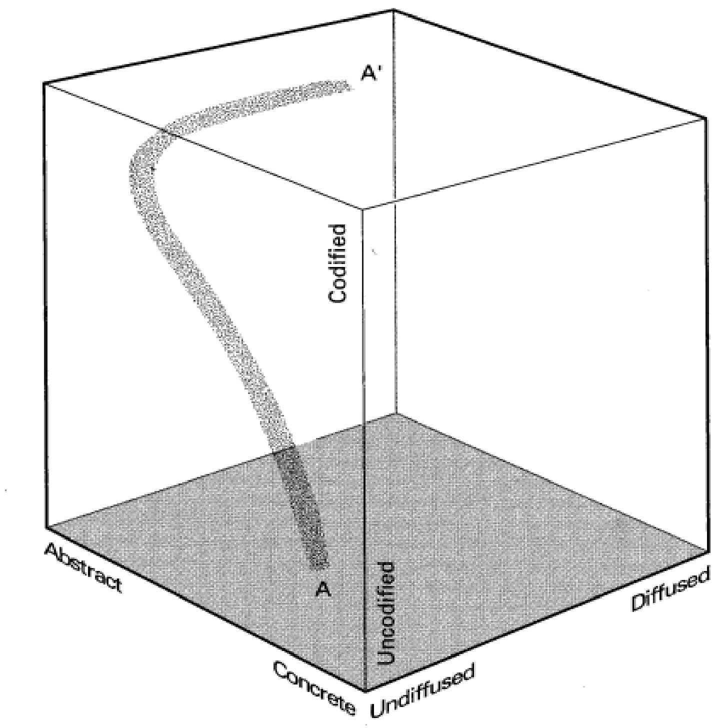


Figure 5.7: The Diffusion Curve in the I-Space

Source: Boisot (1998, p. 56)

Figure 5.7 shows the path of information diffusion in the I-Space along a curve AA'. The figure illustrates the main hypothesis of Boisot's theory, namely that the mutually reinforcing actions of codification and abstraction facilitate the diffusion of information. In the neoclassical approach all information is available without cost (zero information friction), and the population is therefore inhabits point A' on the diffusion curve. Any other point on the diffusion curve implies that information diffusibility encountered some obstacles, and is not friction free[267].

The diffusion curve shown in figure 5.7 represents a static version of the I-Space, but in fact, information flows do not necessarily flow in the direction of the greatest level of codification, abstraction and diffusion. Boisot illustrates the dynamic nature of the I-Space by showing the movement of knowledge in the I-Space.

[267] Boisot, M., H. 1998. *Knowledge Assets* p. 56

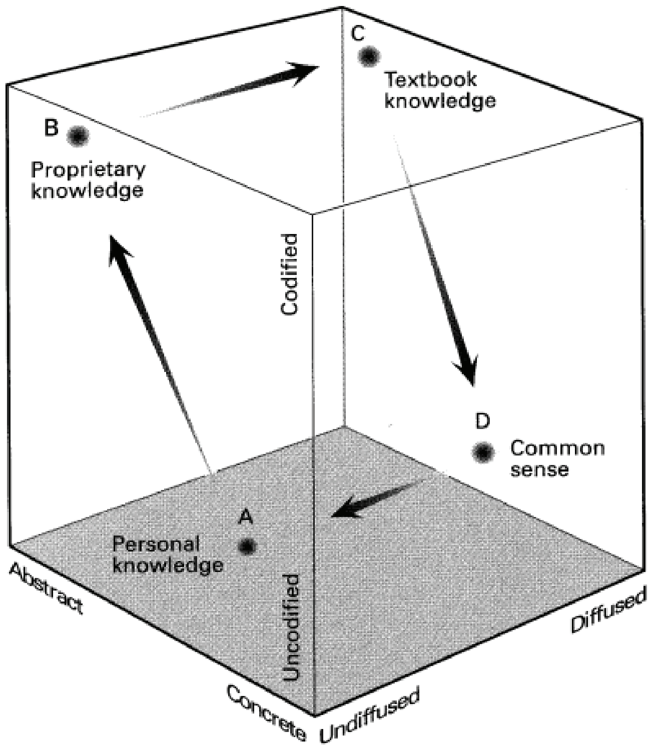


Figure 5.8: The Movement of Knowledge in the I-Space

Source: Boisot (1998, p. 59)

The movement of knowledge in the I-Space is demonstrated in figure 5.8. Knowledge evolves from being based on personal knowledge, to becoming proprietary knowledge (i.e. belonging to a person or institution in the form of a patent or other representation). It then moves in to the space of textbook knowledge – i.e. freely available in a coded and abstract form. At points B and C the knowledge is highly diffusible – even though there may be institutional barriers to diffusion (i.e. legal rights such as copyright or patent limitations), the knowledge can in principle be diffused very easily. In the final stage knowledge is internalised by various agents, and returns to the domain of *common sense*[268]. Knowledge as common sense is not easily diffusible at all – it is internalised knowledge, which is hard to convey to other people. Boisot proposes that the creation and diffusion of new knowledge activate all the dimensions of the I-Space, and that it does so in a specific sequence[269]. This sequence is composed of six phases, and gives rise to a cycle of learning referred to as the *Social Learning Cycle*, or SLC. The six

[268] Boisot, M., H. 1998. *Knowledge Assets* p. 58
[269] Boisot, M., H. 1998. *Knowledge Assets* p. 58

phases of learning that occur during the SLC, are as follows[270]:

- **Scanning –** the act of identifying threats and opportunities in generally available data, of any level of codification and abstraction.
- **Problem-solving –** the process of giving structure and coherence to the results of the scanning process.
- **Abstraction –** generalise the application of the insights codified in the previous step to a wide range of applications
- **Diffusion –** share the insights of the previous steps with a wider, relevant population.
- **Absorption –** apply the codified insights to various situations by means of *learning-by-doing*.
- **Impacting –** the final stage, when abstract knowledge becomes part of concrete practices.

Figure 5.9 illustrates the social learning cycle, showing each of the phases through which the process of creating and diffusing new knowledge travels. Boisot notes that the SLC should be uses as a *schematic* - under different scales, some of the steps may run concurrently.

[270] Boisot, M., H. 1998. *Knowledge Assets* p. 60

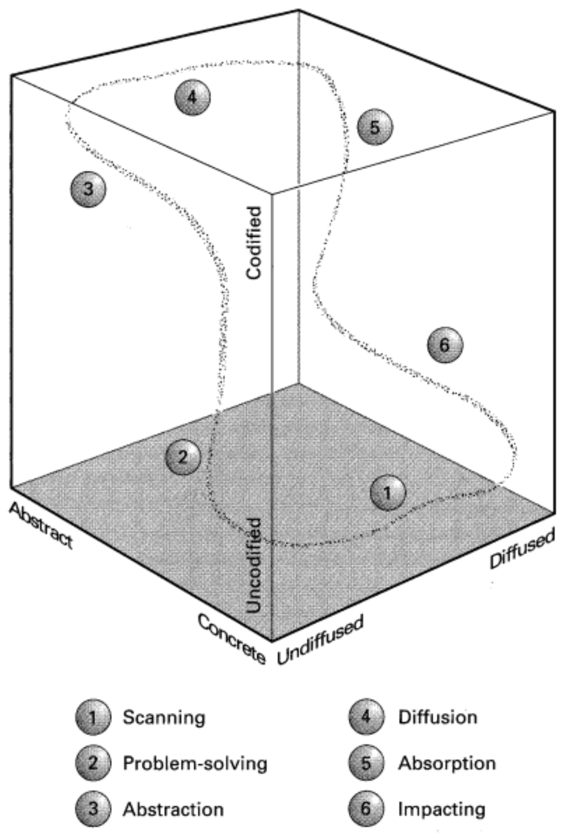


Figure 5.9: The Social Learning Cycle (SLC)

Source: Boisot (1998, p. 60)

Boisot notes a set of propositions that can be derived from the I-Space. These propositions are as follows (illustrated by figure 5.10)[271]:

- Systems act to minimize the entropy generated by data processing activities[272]
- In figure 5.10, entropy production is at a minimum at point E-min, and a maximum at E-max. The SLC links these two regions, thereby *transforming a given population of data processing agents into an engine for putting knowledge to work*[273].
- Organisations seek to escape E-max, the point of maximum entropy, but the dynamics of the SLC forces them to return to it.

[271] Boisot, M., H. 1998. *Knowledge Assets* p. 67-68
[272] Refer to the discussion on Data, Information and Knowledge, for references to the meaning of entropy – both thermodynamic and information entropy is referred to in this context.
[273] Boisot, M., H. 1998. *Knowledge Assets* p. 67

- Somewhere between the zones of maximum and minimum entropy, organisations encounter complexity – a state between chaos and excessive order.
- As an organisation moves towards region E-max, it encounters an area where it is not possible to extract useful information for work. Only data fluctuations can force the SLC forward from this stage. Such fluctuations create far-from equilibrium states – these are stable, discernible patterns that emerge in the region E-max, but which have not yet been subjected to the codification and abstraction process.

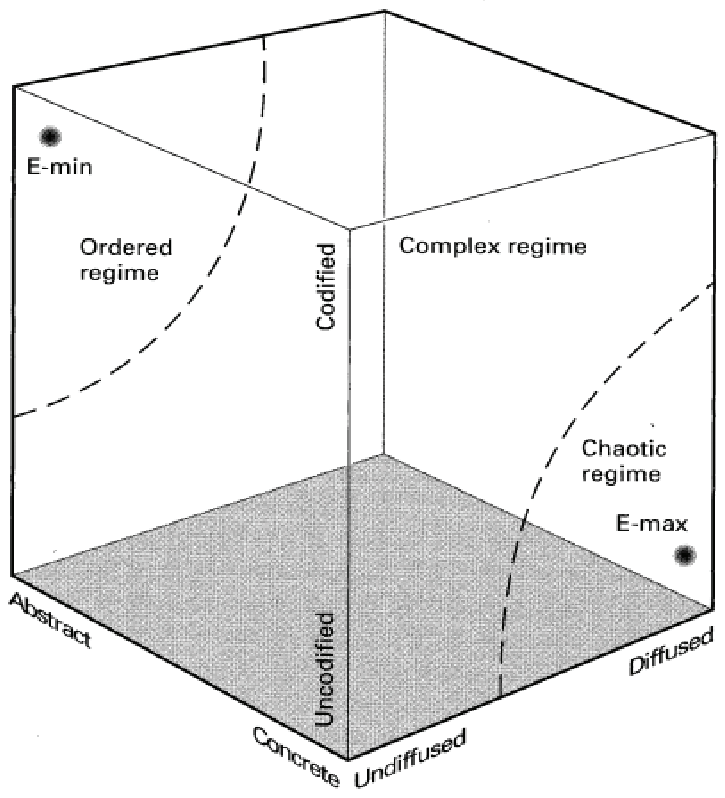


Figure 5.10: Ordered, Complex and Chaotic Regions of the I-Space

Source: Boisot (1998, p. 69)

Given that data is taken to be a factor of production, and that the creation of information offers a means of economizing on this factor of production, then the I-Space offers a framework for analysing the production and distribution of information in a social system at various levels – that of group, firm, or a larger level of aggregation[274]. The I-

[274] Boisot, M., H. 1998. *Knowledge Assets* p. 67

Space will be used to analyse the industrialisation of software production – since we are referring to a a process of industrialisation, the process necessarily takes place at the level of the industry, and the population consists of all those agents involved in the production of software.

In the following section knowledge assets are discussed in detail, using the definitions provided by Boisot and Nonaka.

5.2.4 Knowledge Assets

The aim of this thesis is to substantiate the hypothesis that if the software development industry is industrialized, knowledge assets will come into being in the format of universal production templates. The issue of what knowledge assets are will be addressed in this section.

In the contexts of the research, the definition provided by Boisot will be used:

> *Knowledge assets are those accumulations (of knowledge and information) that yield a stream of useful services over time while economizing on the consumption of physical resources – i.e. minimizing the rate of entropy production*[275].

Knowledge assets cannot be directly observed, in correlation with the idea that knowledge cannot be directly observed – this is because knowledge does not have a physical aspect, but is merely a set of *probability distributions held by an agent and orienting his or her actions*[276]. The existence of knowledge can only be inferred by observing the actions of agents.

Boisot's definition is compared with that of Nonaka:

> *Knowledge assets are the inputs, outputs and moderators of the knowledge-creating proces*[277].

Nonaka does not consider knowledge assets in terms of the impact they have on resource consumption, but simply consider them as assets that have a certain value[278]. He categorises knowledge assets according to the degree of tacitness that the assets exhibit[279]. Nonaka's approach focuses on how the sharing, or diffusibility, of knowledge

---

[275] Boisot, M., H. 1998. *Knowledge Assets* p. 13
[276] Boisot, M., H. 1998. *Knowledge Assets* p. 12
[277] Nonaka, I., Toyama, R., Konno, N. 2000. *Long Range Planning 33* p. 20
[278] Nonaka, I., Toyama, R., Konno, N. 2000. *Long Range Planning 33* p. 20
[279] Nonaka, I., Toyama, R., Konno, N. 2000. *Long Range Planning 33* p. 21-22

assets can be facilitated, rather than on modelling the reasons why this happens[280]. Nonaka does not explain how knowledge assets come into being, but rather provides a strategy for using knowledge assets:

> *Using its existing knowledge assets, an organisation creates new knowledge through the SECI process that takes place in ba. The knowledge created then becomes part of the knowledge assets of the organisation, which become the basis for a new spiral of knowledge creation*[281].

Boisot's definition does not preclude the existence of Nonaka's definition of knowledge assets, since the codified and abstracted knowledge embedded in a knowledge asset may well be an input into the process of further codification and abstraction. Boisot does not address the concept of ba (except insofar ba can be identified with diffusibility), and as such the moderating effect of Nonaka's assets may exist within Boisot's model, albeit not explicitly modelled. The analysis of these definitions suggests that although there is an overlap between the two definitions, they do not refer to the same concepts.

Given the greater explanatory power of Boisot's model in the context of a theoretical approach to modelling organisational knowledge creation, this research relies on Boisot's definition of knowledge assets.

Boisot classifies knowledge assets using a vector consisting of two dimensions, namely codification and abstraction. Codification is a measure of the extent to which a knowledge asset can be given form. Abstraction indicates the extent to which knowledge is more general and less restricted in its scope. The interaction of the codification and abstraction dimensions drives the movement along the diffusion dimension.

Knowledge assets allow us to economize on the consumption of physical resources, and codification and abstraction allow us to economize on the data-processing and communication efforts required to create or use knowledge. *Codification and abstraction lower the cost of converting potentially usable knowledge into knowledge assets*[282]. Knowledge assets are therefore created out of the activities of codification and abstraction.

[280] Nonaka, I., Toyama, R., Konno, N. 2000. *Long Range Planning 33* p. 24
[281] Nonaka, I., Toyama, R., Konno, N. 2000. *Long Range Planning 33* p. 22
[282] Boisot, M., H. 1998. *Knowledge Assets* p. 14

Returning to the earlier distinctions Boisot made between data and information, and information and knowledge, the question arises of how knowledge assets fit into the scheme of conversion from data to information, to knowledge. Boisot answers the question by saying that knowledge assets are *the subset of dispositions to act that is embedded in individuals, groups, or artefacts, and that has value-adding potential*[283]. Knowledge assets act on information and knowledge, adding value in the process.

In terms of the evolutionary production function, knowledge assets are identified as useful, productive structures that are located along an isoquant. Because information economizes on the consumption of data, it may be said that the data-consumption characteristics of these knowledge assets are determined by their information content[284]. In the evolutionary production function, knowledge assets are the outcome of a two-stage process of knowledge creation and application. Creation is the movement across isoquants, and application the movement from right to left along a given isoquant.

Finally – since knowledge flows can be represented in the I-Space, it should be possible to represent knowledge assets in the I-Space. Boisot notes that assets are stocks, not flows. Knowledge assets are stocked in people's heads, in documents and artefacts – a firm's knowledge artefacts can manifest themselves in the firm’s organisation, technologies and products[285]. Since knowledge assets yield a stream of services, one should be able to identify the services that are rendered by the asset. In terms of product, the services are rendered as functionality, reliability and price. Technology realises the services in terms of the efficiency and speed with which a given input is transformed into useful outputs[286]. The services rendered by knowledge assets are expressed at the organisational level by the way that processes and technologies can be configured to respond to changing situations such as demand shifts, competition, regulatory changes, etc[287].

The effect of a knowledge asset on the output of an organisation may be likened to the effects of industrialisation – improved productivity (by means of improved functionality, reliability and price at the same level of resource consumption) and greater manufacturing efficiencies in terms of increased speed of transformation of

[283] Boisot, M., H. 1998. *Knowledge Assets* p. 20
[284] Boisot, M., H. 1998. *Knowledge Assets* p. 26
[285] Boisot, M., H. 1998. *Knowledge Assets* p. 63
[286] Boisot, M., H. 1998. *Knowledge Assets* p. 63
[287] Boisot, M., H. 1998. *Knowledge Assets* p. 64

inputs into useful outputs.

Boisot provides a scaling guide that can be used to position a given knowledge asset in the I-Space, based on its degree of codification, abstraction and diffusion. Boisot acknowledges that this process of positioning a knowledge asset in the I-Space will produce subjective results, but also argues that despite the subjectivity it is possible to glean useful information about an organisation's knowledge assets, because the process assists the organisation in articulating what exactly its knowledge assets are and how they are distributed in I-Space.[288].

5.2.4.1 Assets and Linkages

Boisot points out that the definition of knowledge assets have more than one dimension. Knowledge assets can be classified as both assets, and the knowledge of the linkages between the assets – meaning that an organisation may have very definite assets (such as the knowledge of how to build a certain component), but that it may additionally have specialist knowledge of how to make a number of components work together in the most optimal way. The knowledge of how to integrate knowledge assets forms a knowledge asset in its own right[289].

Thus knowledge assets may comprise technologies and linkages both. The two types of assets – technology based and linkage based – may conceivably occupy different positions in the I-Space. The ability of these assets to occupy different positions in the I-Space is what makes it possible for an organisation engaged in a Schumpeterian[290] style of learning to extract competitive advantage from the social learning cycle.

Knowledge assets that arise out of approaches to industrialisation such as software factories consist of both the schemas and templates used to construct the factory, and the knowledge of how to construct the final product based on the schemas and templates. This knowledge of how to integrate the component parts, constitute a linkage as a knowledge asset.

[288] Boisot, M., H. 1998. *Knowledge Assets* p. 65
[289] Boisot, M., H. 1998. *Knowledge Assets* p. 108- 112
[290] Boisot describes Schumpeterian learning as learning by destruction, based on Schumpeter's notion of *gales of creative destruction.* In the process of Schumpeterian, or S-Learning, parts of new knowledge are disseminated in order to promote the learning cycle, but other parts are retained. S-Learning is therefore a combination of neo-classical learning and *destructive* learning. It both retains and destroys the value obtained from the generation of new knowledge assets (Boisot, M., H. 1998. *Knowledge Assets* p. 99).

In the previous section the model of organisational knowledge creation proposed by Max Boisot has been presented. The next section proposes a reconciliation of the theories of Nonaka and Boisot,.

5.3 Nonaka and Boisot

The process of industrialisation focuses on a move from a craftsmanship based approach to production, to a production process that takes advantage of economies of scale. As we have demonstrated, the process of industrialisation in the software industry is likely to take the form of a move towards taking advantage of economies of scope, rather than economies of scale.

We would like to use Nonaka’s narrative of organisational behaviour for describing the transformation of tacit to explicit knowledge in conjunction with Boisot's analytical framework of the I-Space. The combined approach will enable us to analyse the knowledge assets which are likely to arise from the industrialisation process. Since the consensus is that the software development industry is currently at a stage where it utilises a craftsmanship-based approach, it follows that the process of industrialisation will proceed by the conversion of tacit knowledge to embodied knowledge. This section proceeds to reconcile Nonaka's approach with that of Boisot.

Nonaka's knowledge creation cycle is based in the observed actions of members of organisations. It charts the conversion of knowledge from tacit to explicit and back again, through a succession of stages: socialization, externalisation, combination and internalisation. These stages were discussed in detail at the beginning of this chapter.

The four stages can be roughly matched to the movements of knowledge in the I-Space.

- **Socialization** largely corresponds to the stage of common sense and personal knowledge. It represents the sharing of information amongst individuals. To the extent that sharing information codifies it[291], it represents an initial move towards codification.
- **Externalisation** represents a move towards further codification. It converts tacit knowledge to explicit knowledge. According to Nonaka externalisation can be identified as the key to knowledge creation. Nonaka suggest that the sequential

[291] *The very act of describing an information good) partly transfers it, and the more complete the description, the more complete the transfer* (Boisot, M., H. 1998. *Knowledge Assets* p. 74).

use of metaphor, analogy and model provides the most effective and efficient means of converting knowledge from tacit to explicit[292]. This represents an act of codification – the creation of categories that can be used to code data.

- **Combination** is the process of synthesizing data in the knowledge system. It involves combining different bodies of explicit knowledge[293]. The act of combination suggest codification, but also abstraction, in the sense that categories are being combined, leading to fewer overall categories for coding diverse data. Combination represents both codification and abstraction.
- **Internalisation** is the process of embodying explicit knowledge into tacit knowledge. The resultant knowledge assets will be both minimally diffusible (residing in someone's head as they would be), will also be highly uncodified, and will be applicable to very specific tasks – therefore not abstract at all.

The four stages of the SECI spiral can be said to follow the *typical* social learning cycle – the flow of knowledge proceeds from personal knowledge to more and more codified and abstract knowledge, and eventually back to personal knowledge. The knowledge spiral incorporates the idea that it grows in scale as it moves up through the ontological levels represented by the spectrum of agents from the individual to the group. The reason for this is that the knowledge becomes *crystallized* at higher ontological levels, because of the context provided by the organisation. The increase in scale corresponds very closely to Boisot's diffusibility metric – which ranges from the individual to arbitrarily large populations to whom the data is relevant.

Boisot and Li confirm that the move from tacit to explicit knowledge corresponds to a move from embodied to representational knowledge. With a move toward greater codification and abstraction, embodied knowledge becomes narrative knowledge, and finally gives way to abstract symbolic knowledge[294].

Nonaka and Takeuchi's knowledge spiral implicitly assumes that only tacit knowledge which can be communicated is useful to the organisation[295]. Boisot affirms this by pointing to three different types of tacit knowledge[296]:

[292] Nonaka,I., Takeuchi, H. 1995. *Knowledge Management Classic and Contemporary Works p.* 151
[293] Nonaka,I., Takeuchi, H. 1995. *Knowledge Management Classic and Contemporary Works p.* 152
[294] Boisot, M. H. 2007. *Explorations in Information Space* p. 112 - 113
[295] Boisot, M. H. 2007. *Explorations in Information Space* p. 111
[296] Boisot, M. H. 2007. *Explorations in Information Space* p. 57

- Things that are not said, because everyone understands them, and they are taken for granted.
- Things that are left unsaid because nobody understands them.
- Things that are not said, because even while some people understand them, they cannot articulate them without cost.

According to Boisot, Nonaka and Takeuchi's tacit knowledge is of the last of the types listed above. The transformation of tacit to explicit knowledge incurs a cost. The cost of this move is measured in the amount of data processing that needs to be undertaken to effect the transformation.

In another seeming correspondence to Boisot's framework Nonaka also defines a concept of knowledge assets[297]: *Knowledge assets are the inputs, outputs and moderating factors of the knowledge-creating process.* Contrast this to Boisot's definition, which holds knowledge assets to be the accumulations that act on knowledge and information, and add value to them. In Boisot's schema, the interaction of codification and abstraction facilitates the creation of knowledge assets. Knowledge assets are the outcome of the creation and application of knowledge. In Nonaka's scheme, the knowledge factors are the input into the knowledge creation process. Nonaka's classification of knowledge assets can be mapped according to their degree of codification and abstraction, in a similar fashion to Boisot's positioning of knowledge assets in the I-Space. Nonaka's knowledge assets are used to create new knowledge through the SECI process[298]. This is in contrast to Boisot's notion that the movement through the I-Space is driven by the interaction of the codification and abstraction process, which is in turn driven by the economic imperative to minimise the consumption of data resources.

Nonaka's knowledge assets do not fulfil the same role as those of Boisot – in Nonaka the assets are more closely related to *knowledge*, while in Boisot knowledge assets operate on, and are influenced by information. Nonaka does not make a clear distinction amongst data, information and knowledge. Knowledge assets in Boisot's approach can exist at all levels of the social learning cycle, and perform the economic function of adding value. Different knowledge assets are positioned differently in the I-Space, but

[297] Nonaka, I., Toyama, R., Konno, N. 2000. *Long Range Planning 33* p.20-21
[298] Nonaka, I., Toyama, R., Konno, N. 2000. *Long Range Planning 33* p. 22

the assets themselves are not transformed to create knowledge. In a previous section it has been demonstrated that using the definition of knowledge assets given by Boisot does not preclude the validity of Nonaka's definition, although the two definitions do not refer to the same concept. Boisot's definition will be of greater utility in considering the creation of knowledge assets, and therefore his definition is used in this research.

Nonaka does not seem to be able to incorporate the idea of a dominant design explicitly in his model. It is proposed that the concept of dominant design plays an important role in the process of industrialisation. The explicit analysis of the development of a dominant design which is possible within Boisot's framework, further argues for the wider applicability of Boisot's model to a theoretical analysis of organisational knowledge creation.

Nonaka's SECI spiral does not provide an explanation for the transformation of knowledge from tacit to explicit, and does not treat the knowledge flow from an economic-analytical perspective. By fitting the SECI spiral into the I-Space, it becomes possible to analyse the stages of the SECI spiral using the I-Space, thereby underpinning Nonaka's behavioural analysis with Boisot's theoretical approach.

5.4 Conclusion

This chapter has provided an overview of the foundational knowledge management theories that will be used to facilitate the examination of the hypothesis stated in this thesis.

Nonaka's SECI spiral and the extensions to this framework have been discussed. Nonaka's approach is based on the observation of behaviour within organisations. The SECI spiral demonstrates how tacit knowledge is converted to explicit knowledge to the benefit of the organisation.

Boisot's conceptual framework, the I-Space, was presented as an alternative approach to formulating a theory of organisational knowledge creation. Boisot's approach is based on an analysis of information as an economic factor of production.

It was demonstrated that Boisot's I-Space can incorporate Nonaka's SECI process, and that it provides an explanatory framework for Nonaka's approach. Using Boisot's I-Space does not preclude the existence of Nonaka's conceptualisation of knowledge assets, nor does it preclude the existence of ba.

The next chapter will place the process of software industrialisation in the context of the I-Space. The analysis will take into account both the SECI knowledge creation process and the social learning cycle in the I-Space. It will be shown that the industrialisation process will lead to the creation of knowledge assets in the format of universal production templates for the software industry.

Chapter 6
Software Industrialisation and the I-Space

The hypothesis that is postulated and attempted to be validated in this chapter is the following:

If the software industry is moving towards industrialisation then knowledge assets in the format of universal production templates will come into being.

The chapter presents the concluding arguments of this research, drawing on the conclusions from the preceding chapters. In this chapter the following topics will be addressed:

- The ontological level at which the I-Space framework has to be used to model the process of industrialisation, with reference to the rise of dominant designs and knowledge assets at this level.
- The way in which industrialisation manifests in each of the dimensions of the I-Space
- The effect of industrialisation on the conversion of tacit knowledge to explicit knowledge through the SECI spiral, and the reflection of this process in the I-Space.
- The dynamics of the social learning cycle under the presence of a move towards industrialisation will be considered. Certain technical innovations may shift the diffusion curve in the I-Space, which will in turn affect the trajectory of the social learning curve in the I-Space.

Using the characteristics of industrialised industries, it will be shown that knowledge assets will come into being when industrialisation takes place. It will be shown that the economies of scope which are likely to characterise the industrialisation of software production, will lead to the creation of knowledge assets as a direct result of industrialisation, despite this form of industrialisation being characterised by economies of scope and not by economies of scale. The knowledge assets which are created during the process of industrialisation will be shown to be in the format of production templates, and the universality of the production templates will be shown to be a characteristic of the industrialisation process. Thereby it will be proposed that the hypothesis postulated in this thesis be accepted as theoretically valid.

6.1 The level of discourse

Industrialisation is a process that takes place at a different ontological level than that of the firm. Although the process may have its roots in the firm, and ultimately in the minds of individuals, the efficiencies that industrialisation deliver are available across the industry. Industrialisation is specific to an industry, and the level of the industry defines the boundary for this analysis.

Industrialisation is per definition a process that affects an entire industry. The effects of industrialisation potentially affects all organisations within a particular industry, although it is conceivable that particular organisations may choose not to take advantage of the knowledge assets that have come into being as a result of the process of industrialisation. A case in question is that of artisanal gun manufacturers who produce hand-crafted products to a small number of markets, despite the fact that the weapons manufacturing industry is thoroughly industrialised. Certain organisations may also not have access to these knowledge assets due to legal or infra-structural barriers - this may be the case where a particular country is subject to severe trade restrictions imposed by the country or countries from where the industrialisation originates.

The option to utilise the available technology is simply symptomatic of the meaning of diffusion in the I-Space: diffusion signifies that knowledge can potentially be made available to all parties, but it does not mean that all parties will choose (or be able) to utilise this information.

The ontological level of discourse in this research is that of the industry as a whole, and specifically the software development industry. When the universe is changed to that of the industry, rather than that of a single organisation, it is necessary to examine the impact of this change on the way in which dominant designs arise, and at the way in which knowledge assets are represented. It has been stated in a previous chapter that Boisot has not foreseen any limitations to the level at which the I-Space is applicable, and similarly Nonaka has not restricted the size of the universe, as can be seen from the *spiral of knowledge creation* that is represented in Figure 5.2.

From an economic perspective, when looking at an orthodox production function, movements across isoquants are indicative of technical progress – a movement towards the origin indicates that economizing on resources are taking place, since fewer resources are now required to produce a given output. This is consistent with the definition of industrialisation used in this research – improved manufacturing efficiencies are equivalent to the consumption of fewer resources to produce the same output. The same movement across isoquants in the evolutionary function can show the characteristics of industrialisation when a dominant design arises out of the movement. In the previous chapter it was demonstrated that the development of a dominant design can be identified as a process of industrialisation, because it leads to greater productivity and improved manufacturing efficiencies, as well as standardisation which are features on industrialisation.

Knowledge assets are positioned on isoquants in the evolutionary production function, and can be identified in the I-Space, but good care must be taken to consider the appropriate knowledge assets for the level of analysis[299]. When considering the industry as a whole, the knowledge assets that are taken into account will be those assets that are not domain-specific. This means that assets which are applicable only to developing software for specialised industries, such as the medical industry or the aerospace industry, are not appropriate. The knowledge assets that are appropriate for this analysis are those that can be used across domains of specialisation by the entire software development industry – a case in question being software factories, which propose a methodology and framework that may potentially be applied to software development across all domains.

6.2 Industrialisation in the I-Space

Industrialisation is the process whereby an industry undergoes a process of economic and social change that leads to vastly improved manufacturing efficiencies in the specific industry, as discussed in Chapter 2.

In this research, and based on the work presented in the preceding chapters, industrialisation is identified as being present in an industry when evidence of the following can be found:

- Increased manufacturing efficiencies

[299] Boisot, M., H. 1998. *Knowledge Assets* p. 65

- Increased productivity
- Automation, integration and standardisation

The I-Space framework is contained within three dimensions – codification, abstraction and diffusion. We consider the effect of industrialisation in the I-Space, by considering the way in which knowledge flows in each dimension will be affected, thereby deriving the overall effect in the I-Space.

In this analysis we are considering the I-Space at the ontological level of the industry itself as was discussed in the previous section. The population of data processing agents are organisations working in the field of software production. According to Boisot, the diffusion dimension can be scaled to accommodate such a population[300].

6.2.1 Codification

Codification represents the *creation of perceptual and conceptual categories that facilitate the classification of phenomena*[301]. Codification as a dimension of the I-Space framework was discussed in depth in Chapter 5. The codification dimension in the I-Space represents a dimension that is scaled according to a particular definition of complexity: *the number of bits of information required to carry out a given data-processing task*[302].

According to Boisot, the more codified a task becomes, the less flexible the means of execution becomes, but at the same time the task becomes more amenable to being executed by a machine[303]. Tasks that are executed by machines are automated tasks. Boisot mentions that the process of codification – and eventual automation – may lead to resistance by craftsmen, who resist having their skills reduced to a sequence of instructions that may be automated. He points out that the skills are often valued more for *the identity and status they confer than for the utility they yield*[304]. This corresponds to the notion that an industrialised process will yield more utility (interpreted as productivity) than a skills-based approach.

Based on the effect of automation, industrialised processes may be expected to be highly codified, occupying a corresponding position in the codification dimension of the

[300] Boisot, M., H. 1998. *Knowledge Assets* p. 52
[301] Boisot, M., H. 1998. *Knowledge Assets* p. 42
[302] Boisot, M., H. 1998. *Knowledge Assets* p. 46
[303] Boisot, M., H. 1998. *Knowledge Assets* p. 47
[304] Boisot, M., H. 1998. *Knowledge Assets* p. 47

I-Space.

6.2.2 Abstraction

In the discussion on the formation of dominant designs in the previous chapter, it was pointed out that s*tandardisation leads to performance improvements, which in turn leads to further standardisation.* During this process *product attributes co-evolve, to constrain each other mutually*[305].

Abstraction is the process whereby the number of categories are reduced, in order to *let the few stand for the many*[306]. The degree of abstraction a knowledge asset possesses vary between different assets. The degree of abstraction is reflected in the I-Space by locating a given knowledge assets at a position on the abstraction dimension, which is scaled according to the number of categories that need to be drawn on[307]. The fewer the categories, the more abstract the particular knowledge asset is deemed to be. Abstraction as a dimension of the I-Space was discussed in detail in chapter five.

In chapter one it was shown that the challenges of software development include the challenge of complexity. Complexity in this context is a measure of the resources which must be expanded in developing, maintaining, or using a software product,[308] rather than a measure of systemic complexity. The primary tool used in complexity reduction is abstraction. It was demonstrated that complexity reduction through the use of abstraction is facilitated by means of innovations such as software factories. It was further proposed that software factories may be considered to be a feasible approach to facilitating the industrialisation of software development.

It is therefore proposed that abstraction, as the term is used in the context of software development, is equivalent to the usage of the term in the context of Boisot's theory. Software artefacts will embody high levels of abstraction, and will be positioned at the high end of the abstraction dimension in the I-Space – such abstracted artefacts could be in the form of production templates, even more so, if they conform to so called *dominant designs*. This may also reflect a high degree of standardisation – since category reduction allows one representation to be used in the stead of another, a higher degree of conformity to the dominant design (which may be described as

[305] Boisot, M., H. 1998. *Knowledge Assets* p. 36
[306] Boisot, M., H. 1998. *Knowledge Assets* p. 50
[307] Boisot, M., H. 1998. *Knowledge Assets* p. 50
[308] http://sern.ucalgary.ca/courses/cpsc/451/W98/Complexity.html

standardisation) will lead to economising on data resources. The problem of conformity to which one solution is the creation of software components, similarly reflect the employment of high levels of abstraction, as discussed in chapter 3.1.2.

It has been proposed that software development will benefit from economies of scope, rather than scale during the process of industrialisation. Economies of scope are defined by Greenfield[309] et al as *reducing the cost of solving multiple similar but distinct problems in a given domain by collectively solving their common sub-problems and then assembling, adapting, and configuring the resulting partial solutions to solve the top-level problems*. This definition is another way of expressing the notion of letting *the few stand for the many*[310]. Attaining economies of scope may be said to represent a high degree of abstraction based on the foregoing definition.

It is inferred that the use of abstraction as a measure of complexity reduction in software development, coupled with the rise of economies of scope in software development industrialisation will predispose knowledge assets arising from this process to be positioned towards the more abstract end of the scale in the I-Space.

6.2.3 Diffusion

The diffusion dimension of the I-Space is scaled to indicate how diffusible information is. At the highest end of the scale knowledge can potentially be shared very easily by all who have access to it – the level of diffusibility is in other words very high. At the opposite end of the scale data and information is not available to those who want to use it.[311] The diffusion dimension is scaled according to the proportion of a population that can be reached by information with different degrees of codification and abstraction. The population itself is selected based on the potential relevance of the information and data to the members of the population[312].

In the context of industrialisation, the population consists of all actors (i.e. organisations and other entities) that are involved in creating software. A high degree of diffusion will indicate that the knowledge assets are positioned such that they could potentially reach all the members of the population.

Summarising the foregoing discussion – it has been proposed that an industrialised

[309] Greenfield, J. Short, K. 2004. *Software Factories* p. 159
[310] Boisot, M., H. 1998. *Knowledge Assets* p. 50
[311] Boisot, M., H. 1998. *Knowledge Assets* p. 52
[312] Boisot, M., H. 1998. *Knowledge Assets* p. 52

industry, especially one that benefits from economies of scope such as the software development industry, will have knowledge assets that are positioned at high levels of abstraction and codification. According to Boisot the interaction of abstraction and codification will facilitate the degree of diffusibility of information[313]. Because of the high degrees of codification and abstraction that industrialisation leads to, it is expected that a high degree of diffusibility will be evidenced in the presence of industrialisation. Even though knowledge may not, in fact, be highly diffused, this will be due to artificially imposed obstacles to diffusion (such as the enforcement of intellectual property rights), rather than due to the nature of the knowledge.

Industrialisation represents a move away from crafts-based industry implying a move towards both greater abstraction and codification. The effect of industrialisation is that certain knowledge assets will become available at a region of the I-Space where the diffusibility is at a maximum within the industry, as discussed in the preceding parts of this research. This means that all the agents in the industry will have free access to the knowledge assets in question. Similarly the level of codification and abstraction will be at the maximum that can be achieved within the limiting constraints imposed by Godel's theorem[314]. Nevertheless the knowledge assets arising from the process of industrialisation will not be utilised in exactly the same fashion by every agent – this is because Boisot's analysis leads to the conclusion that *strictly speaking, there is no such thing as common knowledge and there is common information only to a limited extent. Only data can ever by completely common between agents.*[315].

In some sense, all processes in the I-Space that simultaneously maximise the value of all three metrics are processes of industrialisation if the ontological level is set to that of the industry, with individual agents comprised of firms.

In chapter five it was shown that Nonaka's SECI conversion process may be accommodated within the I-Space framework. The next section places Nonaka's SECI process into the context of an industrialised environment.

6.3 Industrialisation and SECI

In chapter five the organizational *amplification* of knowledge during the process of

+Boisot, M., H. 1998. *Knowledge Assets* p. 55
[314] Boisot, M. H. 2007. *Explorations in Information Space* p. 134
[315] Boisot, M. H. 2007. *Explorations in Information Space* p. 40

knowledge conversion between successive ontological levels, was demonstrated using Nonaka's knowledge creation spiral, shown in figure 5.2.

From this it may be deduced that Nonaka's knowledge conversion process can take place anywhere in a continuum of ontological levels, including that of the industry. As such the SECI spiral can be applied at the level of the industry. It is proposed however that the knowledge assets (in the sense that Nonaka defines knowledge assets – i.e. *the inputs, outputs and moderating factors of the knowledge-creating process*) will change as the process of industrialisation takes place. The SECI spiral will still take place at all levels of the organisation and beyond it, but the inputs into the spiral will shift once industrialisation has taken place. It has been proposed earlier that dominant designs are likely to come into being as part of the industrialisation process. These designs (which are knowledge assets since they are located on an isoquant in the evolutionary production function) are hard to shift, and constitute a form of lock in[316]. The inputs into the SECI spiral will therefore be impacted on by these dominant designs, and the outputs of the SECI process will not easily move the industry away from position on the isoquant at any ontological level. Once dominant designs have emerged, innovation is likely to be incremental and not revolutionary any more[317] – this suggests that the externalisation phase of the SECI process will become less influential (recall that Nonaka associates the externalisation phase strongly with the knowledge creation process). Similarly the rise of a dominant design steadily eliminates craft-based skills in favour of mechanistic processes[318], which impacts on the socialisation phase of the SECI spiral. Based on the foregoing, it is proposed that the SECI process will become less effective when an industry has become industrialised.

Industrialisation may further impact the organisational knowledge creation process by affecting the *metabolic pathway* that is traced through the I-Space. The latter may be impacted by shifts in the diffusion curve resulting from technological innovation. The potential changes in the social learning cycle when industrialisation takes place are examined in the next section.

6.4 The Dynamics of the SLC

Increasing industrialisation will cause knowledge assets to move towards a position of

[316] Boisot, M., H. 1998. *Knowledge Assets* p. 162
[317] Boisot, M., H. 1998. *Knowledge Assets* p. 162
[318] Boisot, M., H. 1998. *Knowledge Assets* p. 163

highest abstraction, codification and diffusion, as has been postulated in the previous section. The actual shape of the social learning cycle (SLC) along which the knowledge assets are positioned as the industry matures, will change as the industry evolves, according to Boisot's analyses of the changes in the SLC that took place during the process of industrialising computer hardware design and manufacture[319].

In analogy to the process of computer hardware industrialisation, it is to be expected that the SLC's will evolve from *ones that were deep and concrete and located on the left-hand side of the I-Space to ones that were flat and abstract and located in the upper region of the I-Space*[320]. Such a shift in a social learning cycle represents a shift in knowledge owned and controlled by a group of experienced specialists, to *knowledge that is generic, readily available, and usable with a minimum amount of skill and experience*[321]. This reflects precisely a move from a craftsmanship based industry, to an industry that is industrialised.

The main difference in the process of industrialisation between the micro computer industry and the software development industry is that the computer hardware industry was pushed towards industrialisation aimed at exploiting the benefits available from achieving economies of scale. This point was discussed in chapter 4. In the software industry, economies of scale are not available, but economies of scope are available. It has been proposed that industrialisation can take place even though there are no economies of scale available. If it is assumed that industrialisation will take place, the question is if economies of scope will cause industrialisation to manifest differently than economies of scale does. This is analysed below.

An effect of industrialisation is the appearance of standardisation – in products, processes, or other knowledge assets (including linkages). Standardisation takes place when a dominant design arises, which is stable enough to allow enough time for standardisation to take place throughout an industry[322]. According to Boisot standardisation may either facilitate or prevent concentration in an industry where economies of scale can be achieved[323]. It is proposed that, where economies of scope prevail, *the diffusibility of the knowledge will not be compromised, but since the*

[319] Boisot, M., H. 1998. *Knowledge Assets* p. 161
[320] Boisot, M., H. 1998. *Knowledge Assets* p. 161
[321] Boisot, M., H. 1998. *Knowledge Assets* p. 161
[322] Boisot, M., H. 1998. *Knowledge Assets* p. 161
[323] Boisot, M., H. 1998. *Knowledge Assets* p. 166

necessary knowledge of assets and linkages in combination is still a specialised skill combination, organisations within the industry may be able to locate the assets in the non-diffused region of the I-Space for a longer retention time. It is further proposed that, even if an organisation can maintain these knowledge assets with a low degree of diffusion, the assets will either be diffused or the industry will be monopolised by the particular organisation. Ultimately the presence of economies of scope may impact on the amount of time that is required for innovations to diffuse, but will not change the fact of industrialisation.

The creation of knowledge assets through the process of industrialisation represents a process of technological innovation. Certain kinds of innovation, such as that represented by the microelectronics revolution, can have a seemingly paradoxical manifestation in the I-Space. According to Boisot[324] such innovation may affect the diffusion curve in one of two ways – it may facilitate a move towards codification, abstraction and consequently diffusion (because of the organisation's increased data processing capability), or it may inhibit these same three dimensions because of reduced pressure to codify and abstract since the innovation makes it easier to communicate information. The paradox is resolved by considering the diffusion curve to represent the cost of information sharing, and by showing that the innovation will cause the curve to shift.

It is proposed here that the innovations which are represented by the industrialisation in the software industry will not have a similar effect on the diffusion curve. Because of competitive pressure amongst organisations within the industry, the pressure to codify and abstract will not be reduced – the nature of innovation in the software industry does not lead to the kind of easier communication that reduced pressure in the microelectronics industry. Although it will be easier to transmit codified, abstract knowledge, this will impact only the diffusibility of the knowledge, rather than the shape of the curve – since there is no paradoxical pressure.

6.5 Conclusion

In the foregoing chapter, it has been proposed that industrialisation will be deemed to be present if increased manufacturing efficiencies, increased productivity and automation, integration and standardisation manifest itself in an industry.

[324] Boisot, M., H. 1998. *Knowledge Assets* p. 213

The motivation to industrialise can be found in the propensity of the organisation to invest in the articulation of knowledge, rather than in the accumulation of tacit knowledge (complexity reduction rather than complexity absorption)[325]. This motive, coupled with the economic incentives inherent in improved manufacturing efficiencies, provide reasons for organisations to move towards an industrial rather than a craftsmanship based approach to production.

In the software industry, the physical resources that are involved in the production of software are not the resources involved in producing the embodiment of the software – the CD's, DVD's or other media. These resource requirements are minimal, and form part of the reason why software industrialisation will tend towards economies of scope. In terms of the software, the primary physical resources are the members of the teams writing the software. Considering that the evolutionary production function groups all physical resources (labour and capital) into one, the peculiar nature of the resource mix will not have an effect on the outcome of an industrialisation process. In terms of the orthodox (neoclassical) production function, the overbearing importance of labour as a factor of production will tend to allow very little latitude in terms of substitution in a non-industrialised environment.

It is regarded as probable that the industrialisation of the software industry will benefit from, and be motivated by, economies of scope. In this chapter it has been inferred that in the presence of economies of scope, the diffusibility of knowledge will not be impacted, although certain organisations may gain a benefit from keeping ownership of such assets for longer. Ultimately, however, economies of scope will not be an obstacle to industrialisation.

Given the nature of software development, and the rise of paradigms such as software factories, it is probable that the knowledge assets which will enable the industrialisation of the software development industry will be production templates akin to software factories. Such templates will enable software companies to take advantage of economies of scope by producing multiple product lines based on a single template. The production templates embody highly codified, highly abstract knowledge assets. Because of the codified and abstract nature of these assets, they are naturally highly diffusible in terms of Boisot's framework. Since the production templates require a

[325] Boisot, M., H. 1998. *Knowledge Assets* p. 39

degree of skill to implement (the linkages are knowledge assets as much as the components of the templates) there exists an opportunity for organisations to maximise value by attempting to restrict the diffusion of the knowledge.

The industrialisation process economizes on the consumption of data as much as on the consumption of physical resources – essentially the process of industrialisation is a move from skills to technology. Since technology is more codified than skills, it can be deduced that technology represents a low rate of consumption of data (because codification economises on data consumption) compared to the rate of data consumption of a skills based industry. Given that industrialisation by definition improves production efficiencies, a shift to an isoquant located closer to the origin is implied – such a shift is indicative of improved efficiencies, since any point on the isoquant will produce a given quantity of output for lower levels of all inputs. Industrialisation may be seen as a move towards the origin, i.e. across isoquants and towards the origin, by virtue of the fact that industrialisation is typified by more efficient consumption of both physical and data resources. It has been pointed out in the previous chapter that this behaviour is consistent with the establishment of a dominant design in a given domain.

Using the paradigm of dominant designs, it has been shown that, when the ontological level is set to that of the industry, the coming into existence of a dominant design is effectively equivalent to a process of industrialisation. Although there will always exist more than one knowledge asset on a given isoquant (except perhaps in the limited case of a uniquely dominant design) it follows that in the presence of industrialisation knowledge assets will tend to standardisation – this is because industrialisation is closely linked to the creation of dominant designs, which, as has been proposed earlier, tend towards standardisation. The greater the degree of standardisation, the more generally applicable the knowledge assets are.

The conclusion was reached that industrialisation will tend to create knowledge assets that reside in the area of highest codification, abstraction and diffusion of the I-Space (if the population of agents is the entire industry). This adds additional weight to the universality of the knowledge assets – a high degree of codification and abstraction is equivalent to a high degree of generality.

It has now been shown that knowledge assets will arise from the industrialisation of the software industry. It has been shown that these assets will have the format of production templates, and that they will be of a universal, generic nature. Thereby the hypothesis is deemed to be supported by the theoretical evidence presented.

> ***If the software industry is moving towards industrialisation then knowledge assets in the format of universal production templates will come into being.***

Bibliography

ABRAN, A., MOORE, J.W. (eds). 2004. *Guide to the Software Engineering Body of Knowledge, SWEBOK*. Los Alamitos, California: IEEE Computer Society . ISBN: 0-7695-2330-7

ALBIN, S. T. 2003. *The art of software architecture: design methods and techniques.* Indianapolis: John Wiley & Sons, Inc. ISBN: 9780471468295

ALSHARIF, M., BOND, W. P., AL-OTAIBY, T. 2004. Assessing the Complexity of Software Architecture. *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference:* 98 – 103. DOI: http://doi.acm.org/10.1145/986537.986562

BASKERVILLE, R. et al. 2003. Is Internet-Speed Software Development Different?. *IEEE Software.* 20(6): 70 – 77. IEEE Computer Society.

BECK, K., BEEDLE, M. et. al. 2001. *The Agile Manifesto*. [Online]. Available: http://agilemanifesto.org/. Accessed 7 July 2007

BIRK, A. et al. 2003. Product Line Engineering: The State of the Practice**.** *IEEE Software.* 20(6): 52 – 60. IEEE Computer Society.

BOISOT, M. H., MacMillan, I. C., Han, K. S. *Explorations in Informations Space: knowledge, agents and organization*. New York: Oxford University Press, ISBN: 978-0-19-925087-5

BOISOT, M.H. 1995. *Information Space: A framework for learning in organizations, institutions and culture*. London: Routledge, ISBN: 041511490X

BOISOT, M.H. 1998. *Knowledge Assets: securing competitive advantage in the information economy*. New York: Oxford University Press, ISBN: 0-19-829607-X

BOOCH, G., RUMBAUGH, J. & JACOBSON, I. 1999. *The unified modeling language user guide*. New York: Addison-Wesley, ISBN: 0-201-57158-4

BROBERG, J., VENUGOPAL, S., BUYYA, R. 2007. Market-oriented grids and utility computing. *Journal of Grid Computing*. [Online]. Available: http://dx.doi.org/10.1007/s10723-007-9095-3 Accessed: 19

October 2008

BROOKS, F.P. 1987. *No Silver Bullet: Essence and Accidents of Software Engineering*. Computer, 20 (4): 10 - 19

BUXTON, J. N., RANDELL, B. (eds). 1970. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*, available: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.pdf. Accessed: 19 October 2008

CHARETTE, R. N. 2005. Why Software Fails. *IEEE Spectrum Online*. [Online] available: http://www.spectrum.ieee.org/sep05/1685/1. Accessed: 19 October 2008

CLEMONS, E.K., ROW, M.C. 1991. Sustaining IT Advantage: The Role of Structural Differences. *MIS Quarterly*. 15(3): 275-292.

COCKBURN, A., HIGHSMITH, J. 2001. Agile software development, the people factor. *Computer*. 34(11): 131 – 133. DOI:10.1109/2.963450

COPLIEN, JO. 1994. Borland software craftsmanship: A new look at process, quality and productivity. *Proceedings of the Fifth Annual Borland International Conference*. Available: http://citeseer.ist.psu.edu/346562.html

CUSUMANO, M. et al. 2003. Software Development Worldwide: The State of the Practice. *IEEE Software*. 20(6): 28 – 34. IEEE Computer Society.

DE GYURKY, M. S. 2006. *The cognitive dynamics of computer science: cost-effective large scale software development*. Hoboken: John Wiley & Sons, Inc. ISBN: 978-0-471-97047-7

DEEK, F. P., MCHUGH, J.A.M. & ELJABIRI, O.M. 2005. *Strategic software engineering: an interdisciplinary approach*. Boca Raton: Auerbach Publications. ISBN: 0-8493-3939-1

DRUCKER, P.F. 1993. *Post-capitalist society*. Harper Collins. ISBN: 0-88730-620-9

EISCHEN, K. 2002. Software development: an outsider's view. *Computer*. 35(5): 36 – 44.

ERDEN, Z., VON KROGH, G., NONAKA, I. 2008. The quality of group tacit knowledge. *Journal of Strategic Information Systems*. 17: 4-18

GAMMA, R., et al. 1995. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley. ISBN: 0-201-63361-2

GIBS, W.,W. 1994. Software's Chronic Crisis. *Scientific American*. 271(3): 86-95.

GREENFIELD, J., SHORT, K. 2004. *Software Factories: assembling applications with patterns, models, frameworks, and tools*. Indianapolis: Wiley Publishing. ISBN: 0-471-20284-3

GUAH, M. (ed). 2008. *Managing Very Large IT Projects in Businesses and Organizations*. Chapter 1. Information Science Reference. [Online] Available: http://www.igi-global.com/downloads/excerpts/9653.pdf . Accessed: 19 October 2008

GUSTAFSON, D.A. 2002. *Schaum's Outline of Theory and Problems of Software Engineering.* New York: McGraw-Hill Professional. ISBN: 9780071406208

HAYES, B. 2008. Cloud computing. *Commun. ACM* 51, 7 (Jul. 2008), 9-11. DOI: http://doi.acm.org/10.1145/1364782.1364786

JANERT, PK. 2003. Art, Craft, Science - How about Profession? *IEEE Software*. 20(6): 108 – 109. DOI: 10.1109/MS.2003.1241380

KALISKI, B. S. (Ed.). 2001. *Encyclopedia of Business and Finance*. Vol. 2. New York: Macmillan Reference

KOCH, JAMES C., INMAN, A. R., 2006. Economies of Scale and Economies of Scope, in: Helms, MM (ed). 2006. *Encyclopedia of Management, 5th ed*. Detroit: Gale. 209-211. Gale Virtual Reference Library. Gale. University of Stellenbosch. 1 May 2008. Available: http://go.galegroup.com.ez.sun.ac.za/ps/start.do?p=GVRL&u=27uos

KROLL, P., MACISAAC, B. 2006. *Agility and discipline made easy: practices from OpenUP and RUP.* New York: Addison-Wesley. ISBN: 0-321-32130-8

LANGLOIS, R.N. 1992. External Economies and Economic Progress: The Case of the Microcomputer Industry. *The Business History Review.* 66(1): 1-50

MARASCO, J. 1999. *The software development edge*. New York: Addison-Wesley. ISBN: 0-32-132131-6

MCBREEN, P. 2001. *Software Craftsmanship: The New Imperative.* Addison-Wesley Professional. ISBN: 978-0201733860

MCCLURE, R.M. 2001. The NATO Software Engineering Conferences – Introduction. [Online] Available: http://homepages.cs.ncl.ac.uk/brain.randell/NATO/Index.html Accessed: 23 September 2008

MCCONNELL, S. 2006. *Software Engineering Principles*. [Online]. Available: http://stevemcconnell.com/ieeesoftware/eic04.htm. [10 July 2007]

MESSERSCHMIDT, D. G., SZYPERSKI, C. 2003. *Software ecosystem: understanding an indispensable technology and industry.* Cambridge, Mass.: MIT Press. ISBN: 9780585482651

NAUR, P., RANDELL, B. (eds). 1969. *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*, available: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.pdf

NONAKA, I. 1991. The knowledge-creating company. Reprinted in *Harvard Business Review*. 2007. July-August: 162 -171.

NONAKA, I., REINMOELLER, P., SENOO, D. 1998. The 'ART' of knowledge: systems to capitalize on market knowledge. *European Management Journal* 16(6): 673 – 684. Elsevier Science Ltd.

NONAKA, I., TAKEUCHI, H. 1986. The new new product development game. *Harvard Business Review.* 64(1): 137-146.

NONAKA, I., TAKEUCHI, H. 1995. Theory of Organizational Knowledge Creation. Reprinted in *Knowledge Management Classic and Contemporary Works*. 2001. 139 -182.Universities Press.

NONAKA, I., TOYAMA, R., KONNO, N. 2000. SECI, Ba and Leadership: a Unified Model of Dynamic Knowledge Creation. *Long Range Planning* 33

(2000): 5-34

NONAKA, I., TOYAMA, R.2003. The knowledge-creating theory revisited: knowledge creation as a synthesizing process. *Knowledge Management Research & Practice*. 1:2–10. DOI:10.1057/palgrave.kmrp.8500001

O'NEIL, S. L., HANSEN, J. W. 2001. Productivity. *Encyclopedia of Business and Finance*.Ed. Kaliski, B. S., .Vol. 2. New York: Macmillan Reference USA, 708-710.*Gale Virtual Reference Library*. Gale.University of Stellenbosch.14 Sept. 2008 <http://go.galegroup.com.ez.sun.ac.za/ps/start.do?p=GVRL&u=27uos>.

PLUMMER, D. C., BITTMAN, T. J., AUSTIN, D., CEARLY, D. W., SMITH, D. M., 2008. *Cloud Computing: Defining and Describing an Emerging Phenomenon*. Gartner Research.

PYRITZ, B. 2003. Craftsmanship Versus Engineering: Computer Programming—An Art or a Science? *Bell Labs Technical Journal* 8(3): 101–104. Wiley Periodicals.

RANDELL, B. 1979. Software Engineering in 1968. *Proceedings of the 4th international conference on Software engineering.* 1-10. IEEE Computer Society.

ROYCE, W. 1970. Managing the Development of Large Software Systems, *Proceedings of IEEE WESCON 26(August)*: 1-9.

SANGWELL, K. 2007. Implications of Software + Services consumption for enterprise IT. *The Architecture Journal*. 13 : 18 – 23. Microsoft Corporation.

SHOLLER, D., SCOTT, D. *Economies of Scale Are the Key to Cloud Computing Benefits*. 2008. Gartner Research.

SIRCAR, S., NERUR, S. P., MAHAPATRA, R. 2001. Revolution or Evolution? A Comparison of Object-Oriented and Structured Systems Development Methods. *MIS Quarterly*. 25(4): 457-471

STEPANEK, G. 2005. *Software project secrets : why software projects* fail. Berkeley: Apress. ISBN: 1-59059-550-5

SUTHERLAND, J., 2004. Agile Development: Lessons Learned from the First Scrum. *Cutter Agile Project Management Advisory Service: Executive Update*. 5(20): p. 1-4

SUTHERLAND, J., 2005. *Future of Scrum: Pipelining of Sprints in Complex Projects*. in AGILE 2005 Conference. Denver, CO: IEEE.

WEICK, K.E., 1995. *Sensemaking in Organisations.* Sage Publications. ISBN: 0-8039-7177-X

WOLFE, M. 1955. The concept of economic sectors. *The Quarterly Journal of Economics.* 69(3): 402 – 420.

ZHU, Z. 2006. Nonaka meets Giddens: A critique. *Knowledge Management Research and Practice* 4: 106-115. Palgrave Journals