Doing more by knowing less:

Abstraction and the changing technology of software creation

Magnus Holmén, Rögnvaldur Sæmundsson

www.chalmers.se/tme/EN/centers/ride

RIDE/IMIT Working Paper No. 84426-002

IMIT – Institute for Management of Innovation and Technology
RIDE – R&D and Innovation' and 'Dynamics of Economies

VINNOVA

CHALMERS

Institute for Management of Innovation and Technology

Doing more by knowing less: Abstraction and the changing technology of software creation

Magnus Holmén (corresponding author)
School of Technology Management and Economics
Chalmers University of Technology
Vera Sandbergs allé 8, 412 56 Göteborg
SWEDEN
Email: firstname.lastname AT chalmers DOT se

Rögnvaldur Sæmundsson
School of Business
Reykavik University
Ofanleiti 2, 103 Reykavik
ICELAND
Email: rjs AT ru dot is

Abstract

The paper contributes to the conceptualization of how knowledge grows in the economy by analysing the evolution of mechanisms for economizing cognition in software development processes. These mechanisms are called abstraction mechanisms and they determine what representations, called abstractions, developers can create and use when writing software code. The development of these mechanisms has been an ongoing concern within the software community. We find that over time complementary advances in theoretical knowledge, instrumentation, and computational capacity have led to an expansion of the types of abstraction mechanisms in use. As a consequence software is being composed of a large interrelated network structure of abstractions which have been created by a large number of developers. As the network structure expands and becomes more compact (fine-grained) the ratio between developers' knowledge and the total knowledge they are able to draw upon in their development work becomes lower. They are thus able to do more by knowing less. We suggest that the findings have a general significance for understanding knowledge growth in the economy by furthering our understanding of modularity and the division of innovative labour.

1. Introduction

> [A]ll action is decided in the space of representations…. To explain human actions, both successful and unsuccessful, we often need to understand the representations on which they are based.
>
> (Loasby 1999, p. 10)

This paper aims to contribute to the conceptualization of how knowledge grows in the economy by addressing the evolution of mechanisms for economizing cognition in development processes. The importance of the growth of knowledge for economic growth and development has been widely recognized by economists. Even if the growth of knowledge has been considered outside of the scope of mainstream economic analysis, there is a long tradition, and recent resurgence, of considering knowledge growth as a key driver of economic growth (Smith 1776, Menger 1871, Arrow 1962, Pavitt 1998). Metcalfe (2002) stresses that development processes by economic agents are deeply connected to the growth of knowledge as they create the capacity to transform the economic system from within, maintaining a potential for ever present change.

There are at least two perspectives on the growth of knowledge in the economy. First, knowledge is in some manner being improved, in the sense that problem-solving is becoming increasingly powerful in reaching desired end results given specified starting conditions (Arora and Gambardella 1994). This perspective stresses how humans improve their knowledge of the world and how this improved knowledge increases the ability of humans, firms and other actors to reach desired outcomes.

The second is the topic of this paper that deals with how the total knowledge in the economy grows through differentiation. Through the division of knowledge humans are able to increase total knowledge in the economy above the cognitive limitations of each individual (Pavitt 1998, Loasby 2000). This type of knowledge growth requires coordination in the terms of mechanisms which allow each individual to take advantage of differentiated knowledge of others without sharing that knowledge completely. This perspective focuses on the economising of human cognition. The two perspectives differ in that the former stresses that knowledge is becoming ‘better’ while the latter stresses that knowledge grows in the sense that the reorganization of knowledge allows humans to ignore much knowledge (Loasby (2000, 2001).

The two perspectives do not exclude each other as the growth of knowledge in the economy can and do contain both aspects simultaneously. Thus, the two perspectives are complementary in the sense that improved knowledge may influence the structure of knowledge differentiation (Menger 1871), and knowledge differentiation is conducive to knowledge improvement (Smith 1776).

There are a number of mechanisms that enable economic agents to economize on cognition. Institutions, interpreted as "rules of the game" (North 1990, 1993), can be understood as a broad set of such mechanisms (Nelson and Sampat 2001). Perhaps the best known example is the market price system. According to Hayek (1945/1948, p. 86-87) "[w]e must look at the price system as such a mechanism for communicating information...The most significant fact about this system is the economy of knowledge with which it operates, or how little the individual participants need to know in order to be able to take the right action." This means that market actors can discover and develop profitable opportunities without a complete understanding of their sources (Moran and Ghoshal 1999).

There are mechanisms for economizing cognition that specifically address the needs of developers. One example is technical standards which provide technical specifications, e.g. communication protocols, which developers may adhere to in their work. If many developers follow the same standard, labour is coordinated to some extent, which means that different developers do not need to spend much time or effort thinking about alternative designs or approaches, nor do they have to second guess or adapt to rapidly changing interfaces provided by others.

A related example is the use of modular product architecture, specifying explicit interfaces between modules which define the expected functionality of that module in relation to the architecture (Ulrich 1995). As discussed in the modularity literature, such interfaces reduce the knowledge sharing between developers of different modules necessary for implementing the overall functionality of the product and makes standardization possible (Sanchez and Mahoney 1996, Ulrich 1995).

Regardless of which mechanisms are used for economizing on cognition, they are not natural givens. They are based on representations, which have to be invented and developed (Hayek

1945/1948, Loasby forthcoming). For example, the price system is based on the number system representation which in itself is a mechanism for economizing on cognition (Harper forthcoming). Furthermore, modularity is based on representations and the literature and practice of modularity draws heavily on specific mechanisms that build representations even if these issues rarely are discussed in such a manner (e.g. Ulrich 1995).

The selected and used representations determine what is attainable in terms of differentiation and integration of knowledge in the same sense as representation determines the complexity of a structure (Simon 1996, p. 216). This means that the way representations are created influence the representations' usefulness for economizing on cognition and in terms of the opportunities they provide for knowledge improvements. Thus the means for creating representations have a fundamental influence on how knowledge can be differentiated and integrated and ultimately on the growth of knowledge.

Theories of representation are currently underdeveloped, but theories of computer architectures and programming languages illustrate some of the directions they may take (Simon 1996, p. 133). In particular, programming languages are an example of mechanisms for creating and developing representations called abstractions in software. These languages belong to a larger class of mechanisms, termed abstraction mechanisms, which are integral to the technology of software creation (e.g. Guarino 1979, Shaw 1984). New abstraction mechanisms are constantly being created within the software creation community with the aim to aid developers in decomposing software designs in a way that promotes an effective division of developer work and improves software reuse among developers (e.g. Barnes and Bollinger 1991, Tidd et al 1992, Gamma et al 1995, Mili et al 1995, Fichman and Kemerer 1997). The primary motive of the abstractions and the abstraction mechanism is therefore for developers to economize on cognition within development work.

The purpose of this paper is to analyse the evolution of abstraction mechanisms in software creation and how this evolution has influenced the abstractions being created and used by individual software developers when they create new software. By having this focus we provide an example of the evolution of mechanisms for economizing on cognition in development work and provide a conceptualization which we believe is helpful for general understanding of knowledge growth and the division of innovative labour.

The paper is structured as follows. The next section characterises and contrasts two definitions of abstraction. It also characterizes abstraction mechanisms as technologies of technical development and following that analyses the literature on changes in these technologies. Section 3 analyses how abstractions and the mechanisms for creating them have changed, as well as analysing changes in the technology of software creation. Section 4 four discusses the findings and draws some conclusions. The paper finishes by a discussing implications for further research.

2. Abstractions and the evolution of abstraction mechanisms

There are two common definitions of abstractions (Fellbaum, 1998). The first views abstraction as representation of phenomena in terms of a limited number of elements while leaving out or ignoring others, or the act of creating these representations. A simple example of such abstraction is the representation of an employee by his or her title. Such titles provide information about the activities that a person is expected to perform without giving any details about that person, e.g. name, gender, age, and education, or what means are used to perform the activities. The abstraction here consists of the removal of details, which means that sometimes – but not always - the same title can be used for different persons using different means to perform the activities.

The second definition views abstraction as a general concept formed by extracting common features from specific instances, or the process of forming such concepts. Using a similar example as before the concept of the white-collar worker provides a generalization for workers performing activities which do not require manual labour and who are expected to dress with a degree of formality.

Whilst sounding remarkably similar there are distinct differences between these two definitions. The former focuses on the withdrawal or removing of details. What details are withdrawn or removed are dependent on for what purpose the abstraction is created. The latter, on the other hand, focuses on creating representations which hold true for a group of instances sharing common characteristics. This provides opportunity to generalize statements about phenomena which share common characteristics.

Despite these differences these two definitions are also related. The former view of abstractions is the superset of the latter as suppression of details sometimes is based on the (successful) identification of common elements among different instances. Increasing generality requires the identification of common elements, and the subsequent suppression of details, which serves the purpose of generalization from specific instances. Increasing generality is thus one of the purposes which may guide how phenomena is represented in terms of a limited number of elements and what details are left out or ignored. In other words, withdrawal or removing of details is therefore a more fundamental aspect of abstractions as compared to increased generality. In fact, abstractions that are created with the objective to be general, may in use only turn out to suppress details. In this manner, attempts to generalize

may turn out to be ‘mere’ suppressions of detail. Thus when some domain boundary is passed, the generality of abstractions breaks down into suppression of detail.

Representations have an important role in problem solving (Newell 1969, Simon 1996). Every problem-solving efforts starts with the creation of a representation for the problem which is suitable for the problem solving method used to provide a solution. This amounts to creating an abstraction, i.e. represent the problem in terms of a limited number of elements, which provides a suitable input for the problem-solving method being used to find a solution.

For most of our daily-life problems we are able to retrieve from memory a representation which we have used on previous occasions, adapt it to the situation at hand and apply a known procedure which may include the use of an artefact (Simon 1996). In development work the situation is often different. Development work is concerned with the creation of an artefact or a procedure which can be repeatedly used to solve a particular problem. This artefact or a procedure may be either isolated or belong to a larger system, and the sources of the problems being solved may be related to the “contextual needs of society” (Vincenti 1990, p. 203) or the characteristics of the technologies being used (Laudan 1984, Vincenti 1990, pp. 200-207). When developers create new artefacts or procedures they are often faced with problems which they do not know how to solve. They may therefore need to invent and develop new representations which allow them to solve problems using known problem-solving methods, extend or renew the problem-solving methods in order to reach a feasible solution using a known representation of the problem, or to create both new representations and new problem-solving methods.

The abstractions that can be used by developers depend upon the abstraction mechanisms available to them. Abstraction mechanisms are procedures which enable the developer to specify, implement and use a certain type of abstraction. An example of abstraction mechanisms can be found in Ulrich’s (1995) discussion of modular product architectures. Different types of product architectures are different representations of how products are constructed. For example, when developers create a product that is based on a bus architecture they make use of design procedures for arranging the functional elements and mapping them to physical components in a bus like structure. The implementation, and even the validation, of these design procedures may be partly or wholly automated through the use of Computer Aided Design (CAD) tools. These design procedures which help developers specify,

implement and use the representations on which the products are constructed, are abstraction mechanisms.

Naming is another example of an abstraction mechanism which bears resemblance to the example of employee title mentioned before. Naming is one of the most basic abstractions in software creation and allows the programmer to abstract away from addresses in memory (Guarino 1978). Through its use programmers can use descriptive names when referring to memory locations where certain instructions or data is kept. This solves the problem of bookkeeping when instructions or data need to be moved around in the hardware memory. The problem-solving method consists in using a table that links names to memory addresses. Programming languages supporting naming provide mechanisms that automatically create the table, update it when needed, and translate names to memory addresses. Without these mechanisms it would very cumbersome, or even impossible, to use naming abstractions when creating software.

Abstraction mechanisms can thus be considered a specific class of problem-solving methods which aim at solving problems related to the specification, implementation and use of abstractions. As these problems are at the heart of development work abstraction mechanisms are a part of the technology of development work. When development work is of technical nature this technology has been termed the ‘technology of technical change’ (Arora and Gambardella 1996, Dosi 1988).

But how do abstraction mechanisms evolve? We have already noted that within the software creation community there has been an ongoing search for new abstraction mechanisms which could promote a more effective division of developer work and improve software reuse. Thus, at least in this case, it seems that there always exist problems which developers have difficulty solving given existing abstraction mechanisms. A limiting factor is therefore the ability to invent and implement new abstraction mechanisms to deal with these problems.

We will address these limits and how they influence the evolution of abstraction mechanisms in two ways. First, we analytically investigate the nature of problem-solving methods based on Newell (1969). Second, we review the literature on the changing technology of technical change which is mostly empirical in character. Fortunately, these two approaches provide a

converging picture which we can use to analyse the evolution of abstraction mechanisms in software creation.

Newell (1969) argues that problem-solving methods are characterized by their generality and power. The generality of a method is determined by the scope of different problems the method can solve. A method which can solve a larger set of problems is more general than a method which can solve a more narrow set of problems. The power of a method is on the other hand determined by the method's ability to deliver solutions at all, the quality of the solutions, and the amount of resources required. First, it may or may not be possible to obtain a solution to every problem in the problem domain and methods may even differ in the degree it is possible to know what problems they are capable of solving. Second, problem solving methods may differ in terms of the precision of their solutions or how close they are to optimal solutions. Solutions which are precise and optimal are more useful than others and are therefore of higher quality. Third, problem solving methods may require varying amount of resources in order to reach a solution at all or a solution of acceptable quality.

Newell (1969) also argues that there is an inverse relationship between the generality of a method and its power. This equals saying that for each degree of generality there is an upper bound on the power of a problem solving method and this upper bound decreases with increased generality.

The generality of a problem solving method refers to the range of problems it can be used to solve. In our case, abstraction mechanisms which enable the creation of more general abstractions will have lower upper limits on power compared to abstraction mechanisms which enable the creation of less general abstractions.

The upper limits on power can be viewed as a theoretical limit on the power of an abstraction mechanism. The actual power is dependent on how well the implementation of the mechanism measures with regards to Newell's (1969) three dimensions: the probability of a solution, the quality of the solution, and the amount of resources needed to reach a solution. New and 'better' abstraction mechanism should therefore be more powerful at the level of generality at which they operate.

But what factors will improve the probability of solution, the quality of solution, and the amount of resources needed to reach a solution? As this is not explicitly discussed by Newell (1969) we now turn to studies on the technology of technical change as these provide a means to analyse changes in problem-solving methods.

Arora and Gambardella (1994) provide some evidence which suggests that changes in the technology of technical change are due to complementary advances in three areas: theoretical understanding of problems, instrumentation and computational capacity. Notably, these three areas have been much discussed in the literature (Moore 1965, Rosenberg 1976, Nelson 1992, Nightingale 1998) but as far as the present authors have found, there is far less material available that help us understand how the complementarity of the three areas work out or how they link to Newell's two dimensions. Fortunately, this can be overcome to some extent by referring to studies of the development of science and technology.

The argument for improved theoretical understanding of problems is related to the claim that there is an increased scientification of technological change and thus that advances in scientific disciplines have an important influence on technological change (Meyer-Krahmer and Schmoch 1998). The emphasis here is that through the use of science there are improved attempts to capture and understand the principles that govern physical phenomena. In particular, analysis of patent citations shows that inventors increasingly refer to scientific advances in their patent applications (Narin and Noma 1985, Narin and Olivastro 1992).

While admitting the relevance of the increased scientification of technological change other scholars stress that new technological advancements are prime engines of scientific progress (Gazis 1979, de Solla-Price 1984, Meyer 2000, Rosenberg 1992, compare Smith 1795). Importantly, de Solla Price (1984) argues that advances in instrumentation and experimental techniques have driven and stimulated theoretical advances in fundamental science and innovations. Thus, the argument is that advances in physical artefacts or tools such as instruments may generate new opportunities for knowledge creation, regardless of whether these consist of 'technological knowledge' or 'scientific knowledge'. Thus, instruments can be understood as the capital goods of R&D, meaning that their economic significance comes from allowing researchers or engineers to reduce the costs of solving increasingly complex technical problems (Rosenberg 1976, Nightingale 2000). While many studies used to reside in some formulation of the 'linear model of innovation' where scientific advances precede

technological advance leading to innovation (see Mowery and Rosenberg 1979, Kline and Rosenberg 1986, Rosenberg 1992), more recent studies broadly conclude that scientific and technological advances need to be understood as being mutually dependent (Meyer-Krahmer and Schmoch 1998, Meyer 2000).

An important aspect for the development of instrumentation is improvements in computational capacity (Bell and Gray 2002, Bader 2004). These improvements primarily refer to the dramatic advancement in the 'number crunching' abilities of information technologies. One illustration of this is the so called Moore's law, which states that the number of components per integrated circuit will double in 12-18 months leading to dramatic decrease in cost per component and intra-circuit speeds (Moore 1965). Moore based his prediction on empirical observation in the early sixties but this development has been remarkably stable over time.

An illustration of the complementarity between advances in theoretical knowledge, instrumentation and computational capacity is the area of product development of gadgets based upon electromagnetic waves in the microwave range. Over time problem-solving has shifted from simplified and highly specific but powerful approaches to increasingly general algorithms, such as Finite Element Methods (FEM) that solve problems that defy closed-form analytical solutions. Many of these changes involves the solving of engineering problems by directly applying the fundamental theorem (Maxwell's equations) to develop radar or microwave ovens (Oliner 1984). Such problem-solving procedures would have been impossible just a couple of decades ago because of the complexity of the mathematics given the tremendous demand on computational capacity. However, the shift towards much more general but powerless approaches cannot just be understood from the perspective of advances in computational capacity as the very algorithms themselves have been radically improved. One example includes the reformulation of problems so that the problem becomes computer solvable. Another example is the importance of scientifically proven error estimations of how well a computed solution fits the 'real' solution. This allosw engineers to construct gadgets with controlled behaviour before having tested actual physical products. Hence, the exponential growth in computational capacity together with the advances in theoretical knowledge and instrumentation that have made it economically (if not technologically) possible to apply more fundamental and general approaches as the basis for industrial research.

The complementarity between advances in theoretical knowledge, instrumentation and computational capacity are closely related to improvements in the probability of solution, the quality of solution, and the amount of resources needed to reach a solution. Advances in theoretical knowledge improve the understanding of what solutions are possible and how to reach them given certain assumptions. Instrumentation will determine the quality of the solution, especially to what degree the implementation is automatic and reliable. Finally, computational capacity will greatly influence the amount of resources required to reach a solution.

3. Evolution of abstraction mechanisms for software creation

This section analyses changes in the nature of software creation during the last five decades and investigates how the complementary advances in theoretical understanding of problems, instrumentation, and computational capacity have influenced the evolution of abstraction mechanisms in SWC and how this evolution has effected the abstractions available to software developers. The major changes are the co-evolution of increased software complexity and the increased emphasis and ability of developers to create and use multi-level abstractions from the level of the computer to the level of users problem.

3.1 Use of abstraction in software creation

Software creation consists of activities with the objective to create programming code to run on a computer system. These activities include the capturing of application and customer requirements, system design, programming, testing and software maintenance (Prieto-Dìaz 1990, Bellinzona et al. 1994, Glass and Vessey 1998, Weyuker 1998). The computer systems may range from single microprocessors to parallel computers and large systems of interconnected personal computers or mainframes. Accordingly, the topics and knowledge involved in software creation consists of a range of factors, where the relative importance of these factors varies greatly according to the application and the targeted computer system (Basili and Musa 1991, Glass and Vessey 1998).

The importance of abstractions was discussed by programmers in the 1950s (Jones 2003). This could be expected given that abstractions and models do play a crucial role in all types of engineering. At that time, software played a supplementary role to hardware and software creation was thus viewed as a subfield of existing engineering disciplines. Despite this early

recognition, seminal scientific papers of characterising and using abstractions in *software* were not published until around 1970, especially Dijkstra (1968) and Parnas (1972, 1975).

These and other papers were crucial not just for the explicit identification of abstraction as a fundamental principle of software creation, but also for the rise of modular programming as well as software modularity, which is based on abstraction (Parnas 1972, Brooks 1995). Importantly, in his 1972 ACM Turing lecture Edward Dijkstra argues that “We all know that the only mental tool by means of which a very finite piece of reasoning can cover a myriad cases is called "abstraction"; as a result the effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer” (Dijkstra 1972, p. 864).[1]

Another issue is that at the time there was little explicit recognition that the very notion abstraction could mean at least two different things. This is reflected in the extensive conceptual and semantic confusion in scientific journals and textbooks (e.g. IEEE 1983, Zimmer 1985, Booch 1993). More specifically, one view of abstraction in software was that abstractions could be created as finite pieces of reasoning by keeping essential details but suppressing irrelevant details from the perspective of some purpose. An argument was made that such formulations could be made more powerful and general, ultimately allowing a programmer full control of the software in line with the role of abstraction in other engineering disciplines. However, in practice this implies an emphasis on power of the problem-solving methods at the expense of generality. Indeed, problems that computer scientists can successfully solve by a rigorous approach are often dismissed as ‘toy problems’ by industrialists who argue that these lack industrial significance (Shapiro 1997).

A more pragmatic approach was outlined by Parnas (1972) who argued that abstraction should be viewed as information hiding, where ‘clean interfaces’ among chunks of code allowed for a division of knowledge in software creation. This view emphasises generality of the approach at the expanse of power of problem-solving methods.

[1] At the time, the importance of abstractions were not undisputed. Many industrialists and researchers thought that software developers must have full knowledge and understanding of the entire programming code (Mills 1971, Brooks 1995). However, twenty years later such views are outdated (Brooks 1995, pp. 271-272).

So what this tell us about the nature of abstraction in software? In software creation abstractions permit the representation of phenomena in terms of a limited number of elements while irrelevant details are left out or ignored. What details are included and what are left out depends on the properties or attributes judged to be important for a given purpose (Smith and Smith 1977, Guarino 1978, OMG 2001). The purposes may vary greatly and may influence the nature of the abstractions being created. Some, such as academics dealing with formal verification or fault tolerance, tend to have quite a strict and mathematically based view of abstractions. This is in line with the formulations along Dijkstra. While such attempts have a long history in computer science and are successfully used in some areas, these approaches have failed to diffuse widely. Others, notably practitioners working in commercial environments may view abstractions more in the way of suppression of details or information hiding (for example Parnas 1972). The latter is a more inclusive characterisation that is widely disseminated among software developers. This characterisation covers all aspects of software creation and is thus a better conceptualization of what abstractions are and how they are used in software creation.

Hence, the fundamental importance of abstractions as information hiding is that by abstracting, software developers are able to conceptualize a phenomenon in a simplified way and ignore, avoid, or may simply be unaware of a number of 'messy' details. In software creation these 'messy' details may include the concrete working of a specific software module or application, the operating system or the computer hardware. The use of abstractions is therefore the core means by which software developers decompose a given system specification into modules that can be implemented, analyzed and verified independent of each other (Kiczales 1996, Shapiro 1997, Booch 2001, Jones 2003).[2] Such modularity helps developers to cognitively understand the system, enables different groups of developers to work on different modules and opens up of for the reuse of existing modules (Booch 1986, Prieto-Díaz 1990, Shapiro 1997).

When abstractions are implemented in software they include both the presentation of a simplistic view (often denoted interface) of what functionality a software module provides

[2] While developers strive for complete independence it is difficult to achieve in practice. The degree of independence is inversely proportional to the information needed about the inner workings of the module in order for to be able to use it properly. The information needed in each case is dependent on what abstractions are used as well as the abstraction mechanisms. For example, the abstraction mechanisms provided by object-oriented programming languages provide more powerful means for ensuring independence as compared to older types of languages.

and the details of how that functionality is implemented. The user of an implemented abstraction can be the end-user of the software system, the software developer himself at a later stage, or other software developers (McClellan 1998, McKelvey 1998). In this way, abstractions is a way to create a division of labour over time.

The programming code or modules may be related in a hierarchical manner leading to a multiple levels of abstraction. At a given level of abstraction the intention is that the developer need only to be concerned with what functionality is provided by the lower levels, not how this functionality is implemented. At the same time, the programmer can abstract from other aspects of code on the same level, meaning that programmers abstract (hide information) in the horizontal as well as the vertical domain in relation to the hardware. Vertical abstraction means that a programmer working at a given level of abstraction needs only to write a few statements to invoke a functionality that is implemented in a large number of statements at the lower levels. In fact, as the level of abstraction is increased the ratio between the statements written by the programmer and the statements (instructions) performed at the machine level gets lower. In this way, if the programmer can maintain the same speed in terms of the number of written commands per unit of time, his productivity is greatly enhanced by working on a higher level of abstraction.

In a similar way, when implementing the details at the lower levels closer to the hardware, programmers need only to be concerned with how a certain functionality is implemented and may largely ignore the more application oriented issues, such as to what purpose the functionality is used. They are therefore concerned with developing code that can be used in a large number of different application contexts.

The use of abstraction in software creation is not limited to invoking lower level functionality. Abstractions are also used to delineate modules at the same level of abstraction from the perspective of the hardware. Using abstractions is thus a general way of dealing with complexity by allowing selective attention rather than mastering the ‘whole’ (Simon 1962, Dijkstra 1972, Parnas 1972). The principle of abstraction can be seen as a meta- paradigm underpinning all software creation paradigms, such as different software languages, software architectures or modularity. Indeed, the distinction between different programming language, software architecture or software design paradigms are related to different ideas of how software should be structured (Appleby and Vandekopple 1997), i.e. what mechanisms are

provided that enable software developer to specify, implement, and use abstractions. Notably, these mechanisms have changed greatly over the years. Therefore, in the next three sections we will analyze how complementary changes in theoretical knowledge of problems, instrumentation and computational capacity have changed what abstraction mechanisms are available to software developers. After that we will analyse what effects these changes have had on the type of abstractions developers are able to create and use.

3.2 Theoretical understanding of problems related to abstraction

A central problem related to the development of mechanisms and tools to aid programmers to create useful abstractions is the creation and verification of representation in software.[3] This is the problem of specifying an internal representation of programming code that is appropriate for the purpose and expected functionality of the software being created, and verifying that the implemented representation is correct. For example, how should a given specification be translated into programming code and how can it be verified that the resulting software works as intended?

This problem is non-trivial for three reasons. First, the users of the software who specify what the software should do are seldom able to express their specification in a formal language that is consistent with the programming language being used to create the code (Broadfoot and Broadfoot 2003). Second, it has been mathematically proved that it is impossible to create a general computational routine (algorithm) that can verify if an algorithm completes (successfully terminates) its computation for all possible inputs (Gödel 1931, Turing 1936).[4] Third, testing all possible states that a program can enter is infeasible even for relatively simple programs (Jones 2003).[5] Thus, it is not possible to create general methods which provide a complete validation of representations in software. Instead methods have to be adapted specifically to each case or a set of related cases. This means that the balance between

[3] There are other problems central to computer science and software engineering that are indirectly related. These problems include computation, which is the problem of determining the computational feasibility and creating and analysing computational routines (algorithms). Another problem relates project management, which are related to the managing of software development projects involving a group of people. Important advances have been made within these fields which have helped to raise the level of abstraction, for example by standarizing behaviour. However, these issues are outside the scope of this paper.

[4] This is commonly referred to as the undecidability of the halting problem and was first proved by Turing in 1936. The issues raised by the halting problem are similar to those raised by Gödel's incompleteness theorems, which states that it is impossible to create a complete and consistent axiomatization of all statements about natural numbers (Gödel 1931).

[5] A program with n two-way decision points has 2^n possible paths or states.

power and generality has been an ongoing concern in the software creation community from the very beginning (Brooks 1995, Shapiro 1997). The quest has therefore been to develop tractable ways to deal with the problem rather than being able to provide a formal method for dealing with all possible contingencies (Jones 2003).

This non-triviality has led to an ongoing experimentation to support abstraction with the use of abstraction mechanisms (Embley et al 1995, Monroe et al 1997, Gil and Lorenz 1998).[6] These mechanisms are techniques used to create both sides of the abstraction interface; that is the information needed to use the software module that implements the abstraction as well as the means to provide and verify the functionality of the abstraction for some new purpose or context. This means that the abstraction mechanisms are the core conceptualisations that allow developers to use abstractions in their work (Kiczales 1996, Shapiro 1997, Jones 2003).

To illustrate how theoretical knowledge of abstractions and abstraction mechanisms has evolved, we discuss four major events in the development of programming languages (Guarino 1978). The first step resulted in the development of programming language mechanisms in the 1950s and the early 60s which helped to abstract away the specific workings of hardware. Languages such as FORTRAN included abstraction mechanisms such as naming, which is the ability to use mnemonic names for memory addresses; primitive control abstractions, which include WHILE DO loops; basic data abstractions, which include integer and real variables; and basic abstractions for creating modules, such as subroutines. The importance of these abstraction mechanisms was that they helped programmers to automate many simple, but time consuming and error prone, ‘book keeping’ tasks of programming, many of which related to hardware specific problems.

The second step was the recognition of the importance of user defined abstractions for structured decomposition of specification. By user, we emphasize that the user is a programmer, indicating there is a division of labour within the task of programming. The advantage with the new type of abstraction mechanisms was that other software developers but the ones that created the software language could modify the abstractions. This meant that

[6] The concept of abstraction mechanisms varies in its scope in the literature. How the concept is used in this paper is in line with Gil and Lorenz (1998). Their definition has a relatively narrow scope focusing on the technique of abstracting and does not include the tools/artifacts implementing these techniques nor the underlying theoretical knowledge. Common examples of abstraction mechnisms are the techniques of inheritance and encapsulation used in object-oriented programming.

the division of labour was not just limited to having a program language developer and an application developer but that there is the possibility of an arbitrary division of labour based on user defined abstractions. Structured languages such as ALGOL 68, SIMULA 67 and PASCAL provided mechanisms for user defined data object types and in the early 70s languages were introduced to enforce the correct usage of abstract data types. The introduction of abstraction mechanisms which supported used defined abstractions greatly increased the ability of programmers for expressing abstractions which were more closely oriented to the problem that the program was to deal with. In this way, the nature of the actual hardware could for many applications be ignored as a new structure of abstractions above the hardware could be supported. However, at the same time it became more difficult to automatically verify their correctness. Techniques for checking the use of abstract data types helped to reduce the problem, but did not eliminate it.

The third step was the recognition of the importance of abstractions that could mirror real life objects and processes (Friedman 1989). This understanding lead to the development of object oriented languages such as Smalltalk in the 1980s and C++ and Java in the 1990s. In addition to the ability to mirror real life objects, object oriented languages provided more powerful abstraction mechanisms to ensure and enforce the independence of different modules (objects), such as information hiding, inheritance and isomorphism. Thus, objected oriented languages provided both better assistance for raise the level of abstraction closer to the specification provided by the user and more powerful abstraction mechanisms at given levels of generality.

The forth step is the recent dramatically increased use of scripting languages. These have been around since the 1960s but have been much slower compared to other types of programming languages. Thus, they have not been generally useful until recently. These languages have much less emphasis on strong typing and consequently are much more flexible to use (Ousterhout 1998). Specifically, scripting languages are more useful for integrating existing components or code (‘gluing’) than traditional compiled languages. Therefore, their main usefulness comes in terms of reusing (and abstracting) code, written for example in some object-oriented programming language.

While the theoretical understanding of the importance of abstraction for program design was established relatively early it has taken considerable time to develop and implement better

means for expressing and verifying abstractions. In part, the reason for this is the time it has taken to develop appropriate development tools. We now turn to the development of these instruments and how in turn it has influenced theoretical understanding of problems.

3.3 Instruments and abstraction

A number of different classes of development tools have been created to help the creation and use of abstractions and abstraction mechanisms. Prime examples include compilers that transform human written code into machine readable instructions. Thus, the compiler supports certain type of abstractions through the implementation of appropriate abstraction mechanisms.

Most of the development tools (instruments) are software based, meaning the tools consists of software that is created specifically for the creation of software. We will discuss some important types of instruments.

In order to express abstractions an *editor* is needed.[7] The first editors were mechanical punch card machines where each card represented one line of program code.[8] A code for a program consisted of a stack of cards which were read into the computer and each statement translated into machine instructions by an *interpreter* or a *compiler*. The interpreter or compiler checked if the syntax of the statements fit the grammar of the programming language. If there were errors these were printed on a printer.

In the late sixties the use of console screens for computer output became increasingly popular. The use of consoles made the expression and verification of abstraction much easier and the time between the input of the program and generating an observable output also became much faster than before because of the ability to see the code and the software output on the screen and not just on a printer. The use of the console also enabled the creation of the *debugger*, a program for monitoring and influencing the state of a program while it is being executed. By using a debugger the developer is able to discover errors (‘bugs’) in the program which have to do with the logic of the program, including the implementation of its abstractions, in addition to the syntax errors captured by the interpreter/compiler.

[7] The very first computers were programmed by more primitive methods but we don’t include those here.
[8] See Jones (2005) for an historical account of the use of punch cards.

Editors, compilers/interpreters and debuggers remain the main development tools for programmers, but their nature has changed dramatically. First, there has been a change in the programming languages they support in line with the development described in Section 3.3. Not only have compilers supported more complex abstraction mechanisms, which enable abstractions that are closer to the conceptualization of the user's problem rather than the operating of the computer, but there have also been increased opportunities for providing utilities to help developers create and manage these higher level abstractions. An example of such utilities are graphical diagramming tools enabling developers to visually create and maintain a design in a high level design language, such as the Unified Modelling Language (UML), which can automatically generate programming language statements based on these designs.

Second and also related to the increased used of a graphical user interface, development tools have been created in the form of Integrated Development Environments (IDEs) which makes it easier to relate debugging information to the source code. Additional tools for managing a larger code base have also become a standard feature of an IDE, for example, object browsers and source control systems. The latter is extremely important for managing large software projects involving a large number of developers as it provides various means of version control including mechanisms to mark which part of the code is under revision and monitor interdependencies between different modules.

In addition to changes in basic development tools the amount and functionality of existing code available to programmers has changed dramatically. Typically, this code provides a general-purpose functionality, usually at a lower level of abstraction, without imposing a particular design on the software that uses them. Examples of such code are various application programming interfaces (APIs). A case in point is the Windows API provided by Microsoft's Windows operating system.

Related to the popularity of object-oriented programming, existing code increasingly comes as "frameworks". A framework dictates the architecture and the design of a software system where a software designer need only to be concerned with providing the functionality needed for a particular application. Frameworks allow programmers to work at a high level of abstraction as they are specific to a particular problem domain and assume that a single

architecture will work for all applications in this domain (Sparks et al 1996). For example, in the case of the standard Windows application the developer needs to extend the functionality of certain objects as dictated by the framework, e.g. specify how documents are created and saved but does not have to create the whole structure of a Windows application from scratch[9].

The implementation of complex high level abstraction mechanisms, integration of developing tools using graphical interfaces and the management of a large code base make imposing demands on processing power and data storage. While the use of existing code may speed up the development process it is likely to further increase the demands on computational capacity as it is unlikely to be optimally adapted to the problem at hand, both in terms of size and execution speed. Thus, the changes in instrumentation for software creation are very much dependent on increased computational capacity available to programmers. We now turn to changes in computational capacity.

3.4 Computational capacity and abstractions

In the early days of computing the constraints on the usability of computer systems were related to the cost and capacity of the computer hardware (Friedman 1989). These constraints were related to the processing power of the computer hardware, such as the number of instructions that could be performed per second, the size of the internal memory used to store programs and for intermediate results and external data storage.

Improvements in computer technology moving from valves used in the first computers, through using transistors in the 50s, and to using integrated circuits in the 60s lead to remarkable improvements in computer speed as well as memory capacity. Between 1953 and 1965 the average improvement in the performance of processing units and memory were 80% per year while the reductions in costs were 55% for a given performance level (Friedman 1989). In 1965, one of the founders of Intel, Gordon E. Moore estimated that the number of components per integrated circuit would double every 12-18 months leading to dramatic decrease in cost per component at least to 1975 (Moore 1965). Moore based his prediction, later to be termed Moore's law, on empirical observation in the early sixties but this

[9] Frameworks share many of the characteristics of abstraction mechanisms. They specify an interface in the form of objects that are to be extended in a certain as well as an implementation which is based on the extensions done by the developer.

development has been remarkably stable since. Hence, processing speed and memory capacity have increased exponentially for over 50 years. In order to understand the magnitudes involved the Intel 4004 processor had 2250 components on a single chip in 1971 whereas the Pentium 4 processor introduced in 2003 had 42 million components (Intel 2005).

Increased computational capacity has made software developers less sensitive to various speed and storage penalties related to the use of certain abstraction mechanisms. A case in point is the use of automatic verification. One approach that is used industrially is assertions (Hoare 2003). For most types of software, the speed penalty has become relatively less important compared to other considerations in programming.

The increase in computational capacity of individual chips has further been enhanced by increased parallelism (Bell and Gray 2002, Bader 2004). Parallelism has been implemented through vector supercomputers or through the clustering of scalar processor. In the former case the performance increase is related to the structuring of the computer hardware, whereas in the latter case it is highly dependent on the software used to distribute and synchronize work across different computers in the cluster (Bader 2004). This software is based on many years of research on parallel computing and the creation of abstractions that abstract away the details of parallelism for application developers. The creation of these abstractions are based on abstraction mechanisms that enable developers make a distinction between the specification of functionality and the implementation of the functionality through parallel processing.

While improvements in computational capacity have been crucial for the development of instrumentation (development tools) that are able to implement more complex abstraction mechanisms, further increase in computational capacity through networked processors is dependent on increased theoretical understanding of parallel processing and how it can be represented in software. This theoretical understanding provides the foundation for the creation of abstraction mechanisms that developers are able to use to program at a level of abstraction that need not to be concerned with parallelism. These interdependencies between theoretical understanding of problems, instrumentation and computational capacity provide a good example of how the software community addresses the increasing and arbitrary complexity through raising the level of abstractions with the help of more powerful (and complex) abstraction mechanisms.

3.5 Changes in the creation and use of abstractions

The changes in the abstraction mechanisms made possible by complementary advances in theoretical knowledge, instrumentation, and computational capacity have influenced what abstractions software developers are able to create and use. These changes have transformed the nature of software creation during the last five decades and expanded the scope of its application.

In its early days software creation was focused on the scientific calculations as the first computer systems were created by researchers or military personnel interested in performing complex calculations with great accuracy (Friedman 1989, p. 70). The functionality of these programs was almost entirely governed by mathematical equations used to model physical laws. Thus, the created programming code was relatively short and user interaction with the program during program execution was minimal. The focus was on abstracting away the computer hardware to be able to concentrate on the computation itself. Most of these abstractions were provided by the programming language and used by developers.

Today software systems are often used to manage very complex processes, such as work processes in a large multinational firms or air traffic. Consequently many software developers have created increasingly larger and more complex software with extensive user interaction (Friedman 1989). This has been made possible by the creation of user defined abstraction which have been increasingly able to mirror real-life processes.

The enormously expanded use and applications of software demonstrates its generic nature and "arbitrary complexity" (Brooks 1995, p. 184). Indeed, in principle a computer can compute any "function which would naturally be regarded as computable" (Turing 1936).[10] The large scope – infinite in principle - means that it can be very tempting for software developers to 'overstretch' beyond what is technologically, organisationally, and financially reasonable and feasible at any point in time (Dijkstra 1972). Specifically, the expansion outside the physical applications such as prediction of ballistic curves means that most software developers cannot and do not rely on deterministic procedures such as physical laws. Instead they have to rely on representations of (real life) problems that cannot (easily) be

[10] In effect this means that a computer is a universal Turing machine.

expressed by mathematical models. This means that software problems in general are ill-structured (Newell 1969, Simon 1972).

These characteristics make software systems “arbitrarily complex” (Brooks 1995) in contrast to physical systems whose complexity is structured. The point of this statement is that the structure and coordination within different ‘pieces’ of software is set through human created models, often without being able to rely upon an (hopefully correct) ontology of the workings of nature in terms of physical laws and mathematical or numerical approximations of these laws. This means that for most applications or contexts there are no physical laws governing relationships and interactions between different parts of the software. Instead computers systems are an organization of elementary functional components creating a symbol system which holds a set of symbols and a number of simple processes that operate upon structures of symbols (Simon 1996, pp. 17-22). These symbol systems are characterised by semantic and ontological uncertainty because they are developed by humans and the usefulness of the work of one developer may depend on the understanding and actual implementation of the work of another developer, implying the outcome of work may be unknowable. This means that the relationship and interaction between different parts of software are constructed through social interaction. Therefore, software is subject to unforeseen tensions as it is being used.

Not only is software arbitrarily complex in principle in its nature, but its complexity has also been increasing in line with the extended scope of software objectives. This increased complexity is reflected in the code that the hardware runs in terms of its number of lines of code as well as the fact that the number of professions involved in the creation of software have increased.

The issue of how to deal with this increased complexity has been an ongoing challenge facing the software community at large (Shapiro 1997). Following a couple of seminal NATO conferences in 1968 and 1969 in Germany, the computing community at large considered software creation to be a huge problem (e.g. Naur and Randell, eds. 1969). In fact, the situation was deemed to be so bad that the concept ‘software crisis’ was coined, highlighting the widely diffused (but largely erroneous) perception that hardware costs were dropping while software was increasingly expensive. Large software projects in particular have been shown to consistently be over budget, error-prone (‘buggy’) or delayed. In this sense, software seemed to be very difficult or even impossible to evolve or maintain.

However, these aspects of software do not mean there have been no attempts to address them, nor does it mean there has been no progress. Instead there have been many advances over the years including increases in programmer productivity (Brooks 1995, Shapiro 1997, Ousterhout 1998, Hoare 2003). The development of mechanisms and tools to aid programmers to create useful abstractions and to help raise the level of abstraction has been central in addressing the issue of complexity and the "software crisis" (Appleby and Vandekopple 1997, Booch 2001). Specifically, programming at a higher level of abstraction improves the ability to focus on the application or problem domain, while ignoring all or parts of the detailed working of the lower levels. This is made possible by "galaxies of abstractions" which hold together "societies of collaborating objects" (Booch 2001). These galaxies are a network of user-defined representations of the components of a software application. Some of them may be more generically useful than others, e.g. components provided by object-oriented frameworks, but many of them are specific to the application being created.

The existence of a network of components, however, creates the need, or opportunity, for new types of abstractions (e.g. Zimmer 1985). These abstractions 'glue' together different components. Such abstractions are made possible through scripting languages, which are abstraction mechanisms that are not designed for building data structures and algorithms from scratch, but to connect already existing components (Ousterhout 1998). Abstractions are therefore not only used to abstract away the hardware when performing calculation, but also to creating and re(using) an ever expanding universe of arbitrary abstraction galaxies where each abstraction being created represent a component in the final software.

4. Discussion and conclusion

The purpose of the paper is to analyse how mechanisms for creating representations that allow for economizing cognition evolve in software creation and how that evolution influences the representations being created and used by software developers when they create new software. By doing so, the aim is to contribute to our understanding of how knowledge grows in the economy.

In Section 3, we described how the software creation community has searched for new mechanisms to improve support for the abstractions developers can create and use in their work. This has led to a major expansion of the types of abstraction mechanisms that are available for developers and the abstractions that they support.

The earliest abstraction mechanisms allowed for the creation of pre-defined abstraction types which primarily had the objective to omit details of running the software on a computer. This includes abstraction mechanisms that allowed programmers to ignore many messy details of programming, such as keeping track of where programs and data are stored in physical memory. Essentially, these abstraction mechanisms automated many simple, but time consuming and error prone, 'book keeping' tasks of programming.

More recently created abstraction mechanisms allow developers to create their own abstractions enabling them to specify themselves the separation of 'what' a particular software module does from 'how' that functionality is implemented. This had two important consequences for development work. First, it allowed developers much greater flexibility in structuring their code. Instead of the structure being heavily influenced by the structure of the computer hardware it was possible to create a structure on top of that which bore closer resemblence to the user problem being solved. Software code thus became structured more as a hierarchy where low level representations or abstractions of the hardware were connected to higher level representations or abstractions of the user problem. Second, it made it much easier for developers to reuse software modules made by others. As software structures became more multi-level some of the lower level structures had general applicability for different end-user application. The reuse of software modules has however never become trivial as the assumptions made by the developer creating the abstractions might not be completely shared by the developer using them. Alternatively, their objectives might also

differ so that code is reused for some purpose that it is not very well suited for. These types of problems may lead to abstraction mismatches at different levels which may be difficult to remedy and consequently lead to erroneous results or breakdown when the software is run.

Over time, the consequence of the change in abstraction mechanisms, and the enormous expansion in the number of abstraction they can support, is that software code is increasingly composed of a large interrelated network structure of abstractions which have been created by a large number of developers whose work has mostly been coordinated only through abstraction interfaces. This network structure forms what Booch (2001) refers to as “galaxies of abstractions” which connect representations of individual hardware components to representations of the user problem being solved, but normally through a very large number of intermediary representations. Many of these representations are shared by other software applications.

So what explains the evolution of abstraction mechanisms that has enabled the rise of the abstraction network structure? Abstraction mechanisms that are used by software developers are technologies of software creation. This means that the evolution of abstraction mechanisms is a change in the technology of technical change (Dosi 1988, Arora and Gambardella 1994). As discussed in Section 2 and illustrated in Section 3, the evolution of abstraction mechanisms is underpinned by complementary changes in three domains, increased theoretical understanding, improvements in instrumentation and increased computational capacity. To make the use of complex abstraction mechanisms economically and technically feasible to developers required the development of tools that automated their implementation. These tools, in turn, required dramatic increase in computational power.

Advances in the three domains are complementary because of the relationship between the generality and the power of problem-solving methods (Newell 1969). As discussed in Section 2, the generality of a method is determined by the range of problems that the method can solve. The power of a method is determined by the ability of the method to deliver a solution of acceptable quality within some specified resource and time constraints. Newell argues that there is an inverse relationship between the generality of a method and its power. Thus, for a given degree of generality, there is an upper bound on the power of a method and this upper bound decreases with increased generality. This means that in one extreme we have powerful

problem-solving methods of low generality, and at the other extreme we have general problem-solving methods of low power.

Changes in theoretical knowledge and instrumentation do not have a direct relationship to generality or power, as advances in them can improve both dimensions. However, over time there has been a move towards a higher generality in both instrumentation and methods and procedures. Many of these new tools and methods have much lower power than earlier ones. However, improvements in computational capacity will increase power of existing problem-solving methods and allow new and more general methods with lower power to be technologically and economically feasible. Consequently, in software creation, methods and tools of higher generality will become more powerful with greater computational capacity.

Despite this, in software creation, the power of problem-solving that are used by developers is not very high. As described in section three both high power/low generality and low power/high generality methods are used within software creation. The difference between these two are nicely captured in the different views of the leading authorities on abstraction in software creation. Dijkstra and others promoted the creation and use of powerful methods which would give developers (total) control of the behaviour of the software (Shapiro 1997). This means that the behaviour of the software could be verified and demonstrated to be correct before use. Parnas (1972) on the other hand was concerned with the ability to create flexible representations through information hiding without the need to verify the correctness of software before use (Shapiro 1997). In practice, the latter approach has dominated as the former has proven to be extremely difficult. In the word of Brooks “I dismissed Parnas’ concept as a “recipe for disaster”… Parnas was right, and I was wrong” (1995, p. 272). However, in specific instances where verification of software design is of utmost importance, such as in the space sector, the former perspective is extensively used.

Our results support the argument made by Arora and Gambardella (1994) that changes in the technology of technical change is driven by complementary advances in theoretical knowledge, instrumentation and computational capacity. However, the characteristics of the knowledge development resulting from these changes differ greatly. Arora and Gambardella (1994) argued that knowledge is increasingly being cast in more universal categories which enable developers to bring together hitherto non-connected domains or experiments. In that way developers can apply the same general knowledge to a range of different problems by

selectively combining it with specific, or local, knowledge. However, in our case we find that knowledge is increasingly being cast in a network structure of abstractions and is being drawn upon by developers through the use of an expanding set of abstraction mechanisms. As the network structure expands and becomes more compact (fine-grained) the ratio between developers' knowledge and the total knowledge they are able to draw upon in their development work becomes lower. They are thus able to do more by knowing less.

What does this difference imply? These two types of knowledge development mentioned above were in the introduction characterized as, on one hand, stressing how humans improve their knowledge of the world through more powerful problem-solving methods, and on the other hand, stressing that the organization of knowledge allows individuals to economize on cognition. These two ways of how knowledge develops can be linked to different views about the nature of abstraction. When abstractions are viewed as general concepts formed by extracting common features from specific instances, the creation and use of such abstractions means that knowledge is cast in more universal categories. However, when abstractions are viewed as the suppression of details, the creation and use of such abstractions means that individuals are able to economize on cognition. The former is a special case of the latter, as the extraction of common features always involves the suppression of details.

The outcome of this reasoning is that Arora and Gambardella's (1994) characterization of knowledge development is only partial. Their characterisation of knowledge growth is equivalent to saying that knowledge is *both* becoming more general and powerful. The implications in terms of an increasing division of innovative labour follows directly from such a characterisation. However, a more complete picture is to look at the network of abstractions that are supported by abstraction mechanisms which are both of the high generality/low power type and the low generality/high power type. The structured networks of abstractions provide a different characterization for how developers and firms specialize. The abstraction network structures can be understood as a cognitive foundation for the division of labour at a given point in time. This may then allow for division of labour within as well as between firms.

The other implication of the network structure of abstraction is that it is a cognitive foundation for knowledge reuse among developers. This is a source for markets for technology (Arora et al. 2001) but it is a source of specialization beyond markets. We suggest

that this network structure of abstractions provides the technological governance structure of open source and of open innovation activities in software development.

Another issue involves the experimental nature of software development. Here, the role of failure and problems by economizing cognition explains why the issue of integration and validation of correctness is so important for firms. There are truths to the characterization of knowledge being typed into more universal categories. However, Section 2 and 3 illustrated how the attempts to improve knowledge, to make it more general time and again collaps and fails so that purposeful abstractions that hide inessential details 'deteriorate' into 'mere' information hiding or suppression of detail. We argue that this is a fundamental cause of the experimental nature of software creation and why so many software projects fall behind schedule or fail altogether. Such shifts occur when developer overstretch beyond what the purposeful suppression of essential details can account for. In this way, many improvements in knowledge are illusory and these illusions break down as developers attempt to break new grounds.

At least in the case of software creation, knowledge development by improving knowledge seems to be much rarer than the one which is based upon economizing on cognition, which is ubiquitous. Certainly, the attempts of software scientists and engineers is to make knowledge better, but mostly these improvements are constrained within limited domains of applications and these limitations are only understood to some extent at any point in time. Thus, the development described by Arora and Gambardella (1994) may influence the division of innovative labour under certain assumptions of appropriability, but does not provide a complete understanding of it. This is also confirmed by the modularity literature where it is argued that modular product architectures allows for a division of innovative labour (c.f. Sanchez and Mahoney 1996). The approach taken in this paper to focus on abstraction and abstraction mechanisms is a way to reconcile these different bodies of literature. The former is based on a special case of abstraction while the latter is the more general one.

Additionally, the focus on abstraction mechanisms provide a dynamic approach that is missing in the modularity literature (Brusoni and Prencipe 2001, Ulrich 1995).[11] That is,

[11] The use of abstraction mechanisms is not only confined to software development but involves modularity at large as Ulrich (1995, p. 420) notes: "There have been several attempts in the design theory community to create formal languages for describing function..., and there have been modest successes in narrow domains of

changes in abstraction mechanisms have led to changes in the abstractions being used and, consequently the entire architecture of software code. Recently, in software creation, there are new attempts to modularize software that are based upon new types of abstraction mechanisms that take into account cross-cutting concerns that previous - supposedly modular approaches such as object oriented programming - has been unable to deal with. We suggest that analyses of such dynamic processes using the abstraction-based approach presented in this paper is an interesting venue for further research.

application such as elector- and fluid-mechanical systems and digital circuits... There have also been efforts to create informal functional languages to facilitate the practice of design...".

References

Appleby, D. and Vandekopple, J. (1997) *Programming Languages: Paradigm and Practice*, McGraw-Hill Companies, Second Edition.

Arora, A. and Gambardella, A. (1994). The Changing Technology of Technological Change: General and Abstract knowledge and the division of innovative labour, *Research Policy* 23, pp. 523-532.

Arora A., Fosfuri A. and Gambardella A. (2001), *Markets for Technology: Economics of Innovation and Corporate Strategy*, Cambridge, MA: MIT Press.

Arrow, K. (1962) "The Economic Implications of Learning by Doing," *Review of Economic Studies, Vol.* 29, No. 3, pp. 153–173

Bader, D. A. (2004) Computational Biology and High-Performance Computing, *Communications of the ACM*, Vol. 47, No. 11, pp. 35-32-42.

Barnes, B. H. and Bollinger, T. B. (1991) Making Reuse Cost-Effective, *IEEE Software*, January, pp. 13-24.

Basili, V.R. and Musa, J.D. (1991). The Future Engineering of Software: A Management Perspective, *Computer*, Vol. 24, No. 9, pp.90-96.

Bell, G., and Gray, J. (2002) What's Next in High-Performance Computing, *Communications of the ACM*, Vol. 45, No. 2, pp.91-95

Bellinzona, R., Fugini, M.G. , and Pernici, B. (1995). Reusing Specifications in O-O Applications', *IEEE Software*, Vol. 12, No. 2, pp. 65-75.

Berzins, V., Gray, M. and Naumann, D. (1986) Abstraction-Based Programming, *Communications of the ACM*, Vol. 29, No. 5, pp. 402-415.

Booch, G. (1993) *Object-Oriented Analysis and Design with Applications*, Addison-Wesley Professional

Booch, G. (2001) Through the Looking Glass, Dr Dobbs Portal, www.ddj.com, July 16

Broadfoot, G. H., and Broadfoot, P. J. (2003) Academia and Industry Meets: Some Experiences of Formal Methods in Practice, Proceedings of the Tenth Asia-Pacific Software Engineering Conference (APSEC'03)

Brodie, M. L. (1984)On the Development of Data Models, in Brodie, M. L., Mylopoulos, J. and Schmidt, J. W. (eds.) *On Conceptual Modelling. Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer-Verlag New York Berlin Heidelberg Tokyo, pp. 19-47

Brooks, F. P. (1995) *The Mythical Man-Month – Essays on Software Engineering Anniversary Edition*, Addison-Wesley, Reading-MA

Brusoni and Prencipe (2001) Unpacking the Black Box of Modularity: Technologies, Products and Organizations, *Industrial and Corporate Change*, Vol. 10, No. 1, pp 179-205

Chandra, R. and Sandilands, R. J. (2005) Does Modern Endogenous Growth Theory Adequately Represent Allyn Young? Cambridge Journal of Economics, Vol. 29, pp. 463-473

Connell, M. (2002) Python vs. Pearl vs. Java vs. C++ Runtimes, http://www.flat222.org/mac/bench

Cowan, D. A. and Foray, D. (1997) The Economics of Codification and the Diffusion of Knowledge, *Industrial and Corporate Change*, Vol. 6, No. 3, pp. 595-622

de Solla Price, D. (1984). The Science-Technology Relationship, the Craft of Experimental Science, and Policy for the Improvement of High Technology Innovation, R*esearch Policy*, 13, pp. 3-20.

Dijkstra, E.W. (1968) The Structure of the "THE"-Multiprogramming System, *Communications of the ACM*, Vol. 11, No. 5, pp. 341-346.

Dijkstra, E.W. (1972) The Humble Programmer, ACM Turing Lecture, *Communications of the ACM*, Vol. 15, No. 10, pp. 859-866.

Fellbaum, C. (1998) ed. *Wordnet: An Electronic Lexical Database*, Bradford Books
Fichman, R. G. and Kemerer, C. F. (1997) Object Technology and Reuse: Lessons from Early Adopters, *Computer*, October, pp. 47-59

Friedman, A. L. (1989). *Computer Systems Development*. Chichester, UK: John Wiley & Sons.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, Reading MA.

Gazis, D. C. (1979) Influence of Technology on Science: A Comment on some Experiences at IBM Research, *Research Policy*, Vol. 8, No. 3

Gil, J. and Lorenz, D.H. (1998) Design Patterns and Language Design, *Computer*, March, pp. 118-120.

Glass, R. L. (1988) Glass Column, *System Development*, pp. 4-5

Glass R.L. and Vessey, I. (1998). Focusing on the Application Domain: Everyone Agrees It's Vital, but Who's Doing Anything about It?, *HICSS*, Vol. 3, Vol. 187-196.

Guarino, L. R. (1978). *The Evolution of Abstraction in Programming Languages* (No. CMU-CS-78-128). Pittsburgh, PA: Computer Science Department, Carnegie-Mellon University.

Gödel, K. (*1931)*. 'Über formal unentscheidbare Sätze der *Principia Mathematica* und verwandter Systeme I', *Monatshefte für Mathematik und Physik*, Vol. 38, pp. 173–198. Translated 1962, *On formally undecidable propositions of Principia Mathematica and related systems*, New York: Basic Books

Harper, D.A. (forthcoming). "Economic Pathways to Numeracy". Journal of the Theory of Economic Thought.

Hayek, F.A. (1945/1948). "The Use of Knowledge in Society." In F.A. Hayek (ed.), Individualism and Economic Order (pp. 77-91). Chicago: University of Chicago Press

Hoare, C. A. R. (2003) Assertions: A Personal Perspective, *IEEE Annals of the History of Computing*, April-June, pp. 14-25.

IEEE (1983). IEEE. IEEE Standard Glossary of Software Engineering Terminology, The Institute of Electrical and Electronic Engineers, New York, New York.

Intel (2005). *Moore's Law 40th Anniversary*. Accessed at http://www.intel.com/pressroom/kits/events/moores_law_40th/index.htm on September 15, 2005.

Jones, C. (1996) Programming Languages Table, Release 8.2, Software Productivity Research, Inc., www.theadvisors.com/langcomparison.htm

Jones, C. B. (2003) The Early Search for Tractable Ways of Reasoning about Programs, *IEEE Annals of the History of Computing*, April-June, pp. 26-49.

Jones, D.W. (2005) *Punched cards*. Accessed at http://www.cs.uiowa.edu/~jones/cards/ on September 25, 2005.

Kim, H. (2002) Predicting How Ontologies for the Semantic Web Will Evolve, *Communications of the ACM*, Vol. 45, No. 2, pp. 48-54.

Kiczales, G. (1996) Beyond the Black Box: Open Implementation, *IEEE Software*, January, pp. 8-10.

Klevorick, A. K., Levin, R. C., Nelson, R. R., and Winter, S. G. (1995) On the Sources and Significance of Interindustry Differences in Technological Opportunities, *Research Policy*, Vol.. 24, pp. 185-205.

Kline, S., and Rosenberg, N. (1986). An Overview of Innovation. In R. Landau & N. Rosenberg (Eds.), *The Positive Sum Strategy: Harnessing Technology for Economic Growth*,National Academy Press, Washington, DC, pp. 275-306

Langlois, R. N. (2001) Knowledge, Consumption and Endogenous Growth, *Journal of Evolutionary Economics*, Vol. 11, pp. 77-93.

Laudan, R. (1984). "Cognitive Change in Technology and Science" in Laudan, R. (ed.) The Nature of Technological Knowledge. Are Models of Scientific Change Relevant? Dordrecht: D. Reidel, pp. 83-104.

Loasby, B. (1999). *Knowledge, Institutions and Evolution in Economics*, London: Routledge.

Loasby, B. J. (2000) Decision Premises, Decision Cycles and Decomposition, *Industrial and Corporate Change*, Vol. 9, No. 4, pp. 709-731

Loasby, B. J. (2001) Time, knowledge and evolutionary dynamics: Why connections matter, *Journal of Evolutionary Economics*, Vol. 11, pp. 393-412.

Loasby, B. J. (forthcoming) A Cognitive Perspective on Entrepreneurship and the Firm, *Journal of Management Studies*

Menger, Carl, [1871], *Grundsätze der Volkswirtschaftslehre***. Translated 1981,** *Principles of Economics,* , New York, New York University Press, 1981.

Metcalfe JE (2002) Knowledge of growth and the growth of knowledge. Journal of Evolutionary Economics 12(1): 3–15

Meyer, M. (2000) Does science push technology? Patents citing scientific literature, *Research Policy*, Vol. 29, pp. 409-434.

Mili, H., Mili, F., and Mili, A. (1995) Reusing Software: Issues and Research Directions, *IEEE Transactions on Software Engineering*, Vol. 21, No. 6, pp. 528-562.

Mills, H. D. (1971) Top-Down programming in Large Systems, in *Debugging Techniques in Large Systems*, R. Rustin (ed.) Englewood Cliffs, Prentice-Hall

Mills, H. D. (1975) The New Math of Computer Programming, *Communications of the ACM*, Vol. 18, No. 1, pp. 43-48.

Moran, P. and Ghoshal, S. (1999) Markets, Firms, and the Process of Economic Development, *The Academy of Management Review*, Vol. 24, No. 3, pp. 390-412

Moore, G.E. (1965) Cramming more components onto integrated circuits. *Electronics*, 38, 8.

Moran, P. and Ghoshal, S. (1999) Markets, Firms and the Process of Economic Development, *The Academy of Management Review*, Vol. 24, No. 3, pp. 390-412.

Mowery, D. C., and Rosenberg, N.(1979). The Influence of Market Demand Upon Innovation: A Critical Review of Some Empirical Studies. *Research Policy,* Vol. 8, pp. 102-153.

Narin, F. and Olivastro, D. (1992) Status Report: Linkage between Technology and Science, *Research Policy*, Vol. 21, No. 3, pp. 237-249

Nelson, R. R. (1992) What is 'Commercial' and What is 'Public' about Technology, and What Should Be? In Rosenberg, N., Landau, R., Mowery, D. C. (eds.) *Technology and the Wealth of Nations*, Stanford University Press, pp. 57-71.

Nelson, R. and Sampat, B. N. (2001) Making Sense of Institutions as a Factor Shaping Economic Performance, *Journal of Economic Behaviour and Organisation*, Vol. 44, pp. 31-45.
Newell, A. (1969). Heuristic Programming: Ill-Structured Problems. In J. Aronofsky (Ed.), *Progress in Operations Research, III* (pp. 361-414). New York: John Wiley & Sons.

Nightingale, P. (1998) A Cognitive Model of Innovation, *Research Policy*, Vol. 27, pp. 689-709

Nightingale, P. (2000) Economies of scale in experimentation: knowledge and technology in pharmaceutical R&D, *Industrial and Corporate Change*, Vol. 9, pp. 315 - 359.

North, D. C. (1990) *Institutions, Institutional Change and Economic Performance*, Cambridge University Press

North, D. C. (1993) The Sveriges Riksbank Prize in Economic Sciences in Memory of Alfred Nobel 1993, nobelprize.org

Oliner, A. A. (1984) Historical Perspectives on Microwave Field Theory, *IEEE Transactions on Microwave Theory and Techniques*, Vol. MTT-32, No. 9, pp. 1022-1040.

Ousterhout, J. K. (1998) Scripting: Higher Level Programming for the 21st Century, *IEEE Computer*, March, pp. 23-30.

Patel, P. and Pavitt, K. (1997) The technological competencies of the world's largest firms: complex and path-dependent, but not much variety, *Research Policy*, Vol. 26, No. 2

Pavitt, K. (1998) Technologies, Products and Organization in the Innovating Firm: What Adam Smith tells us and Joseph Schumpeter Doesn't, *Industrial and Corporate Change*, Vol. 7, No. 3, pp. 433-452.

Parnas, D.L. (1972) On the Criteria to be Used in Decomposing Systems into Modules, *Communications of the ACM*, Vol. 15, No. 12, pp. 1053-1058.

Parnas, D. L. (1975) Use of the Concept Transparency in the Design of Hierarchically Structured Systems, *Communications of the ACM*, Vol. 18, No. 7, pp.401-408.

Polanyi, M. (1967) *The Tacit Dimension*, New York: Anchor Books.

Prieto-Díaz, R. (1990) Domain Analysis: An Introduction, Software Engineering Notes, *ACM Sigsoft*, Vol. 15, No. 2, pp. 47-54.

Rosenberg, N. (1976). *Perspectives on Technology*. Cambridge: Cambridge University Press.

Rosenberg, N. (1992) Scientific Instrumentation and University Research. *Research Policy* 21, pp. 381-390.

Sanches and Mahoney (1996) Modularity, flexibility, and knowledge management in product and organization design, Strategic Management Journal, pp. 63-76

Simon, H. A. (1962) The Architecture of Complexity. In *Proceedings of the American Philosophical Society*, Vol. 106, pp. 467-487.

Simon, H. A. (1972) The Structure of Ill-structured Problems, *Artificial Intelligence*, Vol. 4, No. 3-4, pp. 181-201.

Simon, H. (1996). *The Science of the Artificial*. Cambridge, MA: The MIT Press.

Shapiro, S. (1997) Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering, *IEEE Annals of the History of Computing*, Vol. 19, No. 1, pp. 20-54

Shaw, M. (1984) The Impact of Modelling and Abstraction Concerns on Modern Programming Languages, in Brodie, M. L., Mylopoulos, J. and Schmidt, J. W. (eds.) *On Conceptual Modelling. Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer-Verlag New York Berlin Heidelberg Tokyo, pp. 49-83

Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M. and Zelesnik, G. (1995) Abstractions for Software Architecture and Tools to Support Them, *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, pp. 314-335

Smith, A. (1776), *The Wealth of Nations.* Dent (1910): London.

Sparks, S., Benner, K. and Faris, C. (1996) Managing Object-Oriented Framework Reuse. *Computer*, September, pp. 52-61.

Tidd et al (1992) The Commodification of Application Software, *Industrial & Corporate Change*,

Turing, A.M. (1936). "On computable numbers, with an application to the Entscheidungsproblem." In *Proceedings of the London Mathematical Society*, 2d series, 42, pp. 230-40.

Ulrich, K. (1995) The Role of Product Architecture in the Manufacturing Firm, *Research Policy*, Vol. 24, pp. 419-440.

Vincenti, W.G. (1990). What Engineers Know and How They Know It. Analytical Studies from Aeronautical History. Baltimore: John Hopkins University Press.

Weyuker, E. J. (1998) Testing Component-based Software: A Cautionary Tale, *IEEE Software*, September/October, pp. 54-59.

Whitehead, A. N. (1911) *An Introduction to Mathematics*, New York, Henry Holt and Company.

Young, A. A. (1928) Increasing Returns and Economic Progress, *The Economic Journal*, Vol. 38, No. 152, pp. 527-542

Zimmer, J.A. (1985) *Abstraction for Programmers*, McGraw-Hill, New York, New York