CHAPTER FOUR

SOFTWARE DEVELOPMENT PRACTICES, INFORMATION SYSTEMS AND ORGANIZATIONS

4.0 Introduction

Chapter 1 gave a roadmap for this thesis. Chapter 2 discussed the philosophical groundings that guided the choice of the research methodology that was later discussed in Chapter 3. In this chapter, the concepts of software development, software development process, software development methodology, mechanistic products and romantic products that were briefly discussed in Chapter 1 are further explained in the context of their use in software development projects. In addition, this chapter also looks at the theoretical groundings, that is, the activity system, actor network system and the theory of organized activity that inform us of the world view that we should subject our organizations and their organizational information systems to.

A software development process is required to produce a piece of software. The software development process can be viewed as a framework or structure that is used during the development of a software product. Some examples of the software development processes are the waterfall, V-model, iterative and spiral models as discussed in Schach (2005) and Pressman (2005:79-88). The software development process prescribes the activities and techniques at times together with the tools that must be used to accomplish the different software development activities. There are varied views to what is software development. For example, Kirlidog and Aytol (2010) and Brooks (1995) view software development as a piece of creative art. This implies that some traits cannot be automated as they heavily rely on the developers' personal characteristics. On another note, Hirschheim *et al*. (1995) regard information systems development (ISD), as the use of information technologies in solving organizational problems. Software development is therefore part of ISD since software products are needed to implement information systems. This process is very complex and "cannot be captured in some formal model" only but requires the inclusion of some social approaches in its solutions (Hirschheim *et al.,* 1995:2).

In terms of players, the software development field is characterized by two major groups of stakeholders: the software developers and the method engineers (Gonzalez-Perez & Henderson-Sellers, 2006). Software developers may be any one of the following: system

analysts, programmers, business analysts or system architects. On the other hand, method engineers define and prescribe the methodology to be used by developers in their quest to construct software products. It is important to note that this researcher is assuming the role of a method engineer by attempting to develop a new software development approach. These two terms, that is, software developers and method engineers, refer to the roles played by the individuals or organizations involved in the software development process.

On the other side of software development, there are also information systems. These consist of three parts: the formal, informal and the technical parts. In current development approaches, the software product is usually part of the formal part of an information system. It is the software products, together with the hardware part of the technical subsystem, that are usually tasked with the day–to-day running of information systems. The formal subsystem has a bureaucratic nature. In it, form and rule replace meaning and intention.

Knowledge is a key factor in the development of software products. This knowledge is found in the organizational structures of the system to be developed. Any information system development methodology, therefore, should be able to transfer this knowledge and share it with the actors in the organization. In addition, it is only in a practice that this organizational knowledge is found. Strictly speaking, the knowledge is embedded in the situations where people perform a practice. For this reason, since information systems should capture and share this knowledge, the tacit nature of the knowledge makes it difficult for system developers to explicitly understand the task of software product development. This chapter will discuss some issues that limit the capturing and transfer of knowledge in organizational information systems.

Descriptively, the informal subsystem consists of a sub-culture in which meanings are established, intentions understood, beliefs are created and commitments and responsibilities are made, altered and discharged. This constitutes the tacit and implicit rules, procedures and power structures that are inherently a permanent attribute of the human factor. The informal subsystem has never attracted much attention from software developers.

In the end, since the informal subsystem is very fuzzily defined, the technical subsystem is used to automate the formal subsystem only, which is only a part of the whole system. This chapter starts with a brief excursion into the nature of organizations and information systems. It briefly addresses the influence of culture, concepts and context in an organization’s

practice. The practice of software engineering is then discussed, focusing mainly on the dictates of the definition of engineering in the field of software development. Together with the theory of organized complexity (TOC), systematicity and system formation, this is used to explain why current software development methods result in mechanistic products.

4.1 The Nature of Organizational Systems

There is currently a major difference between the nature and representation of organizations. In an attempt to match and reduce the gap between these two, three theories, the theory of activity systems (AT), actor-network theory (ANT) and the theory of organized action (TOA), are used to assess organizational systems. This assessment is used to support the argument that organizations and information systems have a socio-technical nature. These theories all together will combine to motivate for romantic world view which is the theoretical grounding for this study. As discussed earlier, sound theoretical and methodological foundations are at the core of a sound research investigation. A theoretical framework should be conceived as a structure that holds together and supports theory of the study. It must be able to explain why the problem exists and must justify the need for research. It is from these theoretical groundings that a summary framework of what makes up a development approach will be deduced.

People, information systems and the environment in which organizations exist can be considered holistically as a social group of actors interacting through networks. As Ngwenyama and Lee (1997) contend, being social involves the alignment of an individual's actions to both the organizational context and the other actors involved in performing a social action. All social interaction is governed by a social culture. This culture has to be observed and studied during system development.

4.1.1 The Complex Nature of Organizational Systems

Organizational systems are examples of dynamic and complex systems. The complexity of these systems can be measured using the concept of requisite variety (Rosenkranz & Holten, 2007:57). Requisite variety views organizational systems as possessing several possible states, in terms of "patterns of behaviour" or "number of manifestations". During software development, it is the developer's intention to capture and maintain these patterns of behaviour (manifestations) in the resultant software product. However, during the software development process, the tasks of modelling, that is, of developing the analysis, design and

implementation models tend to reduce the complexity of these organizational systems by reducing their requisite variety. This, in turn, reduces the possible behavioural states of the subsequent software products and information systems under development. This process is regarded as the reductionist principle. Reduction in the possible behavioural states and, hence, in the requisite variety of the original system, in turn reduces the life responsiveness of the modelled and developed system. It is this researcher's opinion that most information systems fail to provide value to their organisations because of this reduction in requisite variety.

In order to maintain the requisite variety of organisational systems and to transfer them to the developed systems, two fundamental things have to be done. Firstly, either the modelled system has to have its requisite variety reconstituted to the original un-modelled state of the organizational system and, secondly, the original system should never be modelled using the reductionist principle. An alternative would be to allow the implementing tools of the system, as well as the system users, to possess as much requisite variety as that which existed in the original un-modelled system (Rosenkranz & Holten, 2007). Practically, however, it is impossible to have tools and users that have the same behavioural modes as the original system. In the end, the only practical way is to find methods and tools that will maintain the requisite variety during the process of system development. However, Lemmens (2007:57) notes that the failure to develop tools that maintain system behavioural characteristics has curtailed the development of systems that can capture human aspects. This is one of the several research goals of this study.

4.1.2 Culture in Organizations

Another aspect that contributes to the requisite variety of organizational systems is culture. Organizational culture comprises the attitudes, experiences, beliefs and values of people in an organization. It also embodies the organisation's interactional behaviour with its stakeholders. All organizations are run within certain cultural boundaries. As the definition of this culture is difficult and complex, its existence in organizations makes each organization different. Dahlbom and Mathiassen (1993) contend that, in order to understand a culture, one has to observe it in a practice, preferably by participating in that practice.

The information and knowledge that is required in organizations is always intertwined and embedded in a culture. Knowledge, like culture, is also interwoven in the same practice in which culture is observed. Practice is normally dependent on a definite situation. The fact that

knowledge is embedded in a practice, in the situations in which people perform that practice, makes the task of software developers very uncertain. In order to understand the situatedness of the organization, software developers should strive to acquire the operational knowledge of the organization. This knowledge can be gained by developers participating as workers or by means of formal and informal discussions with the workers and all other stakeholders in the organization. Without that, it is difficult to force the knowledge to be applicable to a definite situation such as an organizational information system.

Because of the need to capture organizational culture and knowledge in the organization, system development, hence software development may take several months or years to complete. In practice, therefore, and in the interest of time and project implementation schedules, many projects are continued before the developers have fully grasped the practices of the users.

Another problem in practice is the fact that many software practitioners develop software products for organizations in which they are not employed and have only limited knowledge of the organization's practices. The development of systems for these organizations without software developers having studied their practices forces them to chart knowledge in their heads, knowledge that is not applicable to the organization's culture. In consequence, they develop systems that depend on a practice that is not aligned to the current organization. The systems developed, therefore, portray a non-existent practice. To aggravate this problem, in practice, developers also rely heavily on the use of explicit procedures. As a result, bureaucratic or rule-based systems are developed.

In learning about a practice or when participating in a practice, people in organizations communicate using concepts. The role of concepts in organizations is discussed in Section 4.1.3 below.

4.1.3 The Role of Concepts in Information Systems Development (ISD)

"*I know you believe you understood what you think I said, but I am not sure you realize what you heard is not what I meant.*"

Pressman, 2000.

The above statement highlights one of the most misunderstood and neglected tasks in software development, that is, understanding the meaning of concepts used during

communication. These concepts may be business, domain or technical concepts. The development of concept negotiation techniques is a task that has not attracted enough effort in many software development approaches. Concepts are generally used as communication signs in organizations. The success of a development team in building a language community with all the stakeholders in the development process and, hence, in communicating concepts in the domain effectively, is regarded as one of the success metrics of software development. A language community is created when all the stakeholders involved can communicate and share their knowledge of a system with a common understanding. The use of concepts and their effective communication is of the utmost importance during the domain analysis stage. At this point, a study of the area to be serviced by the software product is carried out. This process requires system analysts to engage with business analysts, system architects and users so that the business environment can be established and understood before the process of system requirements gathering is commenced. Of course, this process also continues into the requirements-gathering stage.

Sowa (1976) characterizes a concept as an undefined primitive. In contrast to this, Buitelaar *et al.* (2003) argue that it is not always easy to identify a linguistic term as a concept. They contend that concepts do not exist or are not a given outside a specific domain. This rebuts Sowa's assertion that anything that a person can think of can be regarded as a concept. Aristotle, however, argued that concepts can be composed of other concepts that are broken down to a set of primitive elementary concepts. In this way, concepts can only be understood by enumerating their primitive elements.

In another discussion, Dahlbom and Mathiassen (1993) seem to agree with Buitelaar *et al.* (2003) that concepts are defined by people's practices. Concepts should satisfy three basic requirements: they should have an intensional aspect, a set of the concept's instances (its extension) and a set of linguistic realizations. Linguistic realizations refer to the multilingual terms that are applied to the concept (Buitelaar *et al.,* 2003). Concepts, therefore, should be defined in a particular context.

In fashioning systems and software products, it is important to note that these should be able to support people in the use and communication of concepts. As concepts are important communication tools, they are, therefore, the medium by which organizational information systems pass on information. These can be likened to obligatory passage points (Introna,

1997). Since each concept is defined within a domain, the role of context in assigning meaning to concepts should be discussed.

4.1.4 Context in Information Systems

As discussed above, concepts apply to a specific practice within a certain context. Context can be regarded as the environmental setting or the circumstances that dictate the occurrence of an event or the meaning associated with it. Furthermore, Ngwenyama and Lee (1997) believe the existence of organizational context serves as a framework for meaningful social actions. This was evidenced in their study of communicative messages, when the continual exchange of messages using emails among actors, eventually led to the building up and better understanding of context. Roque *et al.* (2003) stress the need for context to be understood if one is to successfully develop a software product artefact. They contend that the software artefact should be fashioned in such a way that it fits the context in which it will be used.

Also, organizational communication requires a language to transmit information between actors. In order to enable the actors to understand each other, this language should be entrenched within the context of the organization (Malinowski, 1923). Messages that are communicated in an organizational system are also situated in that particular context. Information systems should thus be tuned in such a way that they capture this organizational context. Capturing organizational context subsequently captures meaning, a very important facet needed to enable organizational actors to communicate.

These actors should also have "full knowledge of circumstances" if they are to have perfect explanations of the situations in which they find themselves (Roque *et al.,* 2003:111). However, it is most unlikely that any actor will gain complete understanding (or full knowledge) of a situation. The reason for this is that knowledge is usually modelled around some limiting factors that make knowledge worthwhile.

Context-building amongst organizational actors is a process of weaving together the different situational understandings of different actors, establishing threads of common understandings and of inter-subjective knowledge within the network (Goldkuhl, 2002; Dilley, 1999). Contrary to what many people think, context is not a static thing and is not a given. It is not self-evident in a situation but requires a constructive machinery to mould the varying situational meanings into a common understanding. Context, therefore, is an object of study that requires some analysis to arrive at an agreed and shared understanding. Furthermore, it is

within the shared meaning of some situatedness that the said context resides. Organizational context is ever changing and is continuous.

This dynamism in context poses a big challenge to software development if one has to capture the running context of an organization. Many software development processes lack methods for identifying, capturing and communicating context in the resultant software product. It is recommended that a methodology or a process be devised for building software and for changing it as and when the context changes. Without the presence of concepts that are communicated in a particular practice and without that practice being defined in a specific context, meaning cannot be communicated in any organizational system. Paradoxically, shared understanding or meaning is the basis for building a situational context. It is, therefore, of paramount importance that software development practices use a methodology or approach that captures this situational context.

Although it is agreed that software products should capture organizational context, there still remains a problem in the software development process of where, when and how this context can be captured, communicated and stored. Although currently, some available software development tools can capture the context, it is difficult to include this context in the software products developed.

This is compounded by the practice of software product development that has always been guided by the need to '*engineer*' a product for use in industry. The use of the word engineering has restricted the focus on culture and context of organizations to such an extent that only those organizational aspects that can be formalized and, therefore, engineered, are implemented using software products. The impact of forcing the definition of engineering on to the software product development process is discussed in the next section.

4.2 The Practice of Software Engineering

This section highlights some problems to software development that have been caused by the use of the word and philosophy behind 'engineering'. Several publications (Schach, 2005; Pressman, 2005; Heineman, 2000; IEEE, 1990) have given software engineering (SE) a plethora of definitions but all of them borrow their core meaning from the definition of the word "engineering". The American Engineers Council for Professional Development (ECPD) defines engineering as:

"*The creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behaviour under specific operating conditions; all as respects an intended function, economics of operation and safety to life and property.*"

Engineers Council for Professional Development (ECPD).

The underlining is for the researcher's emphasis only. Focusing on the underlined phrases, we note that adherence to '*scientific* principles' in software development has limited software developers' attention to those aspects of software development that can be repeated and reproduced. This notion, therefore, disregards the softer elements of organizations such as culture and context that are usually tacit in nature and may not be repeatable. Coupled with the need to '*design'* software artefacts, that is, to model and devise a representation that fits certain constraints, the process has actually limited the requisite variety of software products. In other words, design limits or restricts the variety of operating environments or of possible states in which a software product can be applied and used.

Current software development design processes have limited the ability of software products to be adaptable. This process forces software developers to concentrate on the '*intended function'* of the software product. In organizations, however, such intended function is not a single, static or discrete function, but is dynamic, and continuously changing. In addition, the intended function has a running context that requires its capture and inclusion in the software product. There is, therefore, a limit to the extent to which developers can elicit the intended function of a living system such as an organization. The organization is considered as a living system on account of the fact that humans are always central to its existence.

Also in line with the ECPD definition of engineering, the IEEE Computer Society (1990) defined software engineering as the "*application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software*". The underlined text is for the researcher's emphasis only. As a model, Schach (2005:16) noted the need for developing "fault free software, delivered on time, within budget, and satisfying user's needs". As a framework, Pressman (2005:34) describes SE as "a process, a set of methods, and an array of tools". Adding to that, Heinemann (2000) regards software engineering practice as a field aimed at developing

reliable and robust software products. All these definitions have had a very profound influence on the way software products are developed in practice. Their effect has been to curtail the inclusion of behavioural human aspects of organizations in software products.

Conford and Smithson (1996:11) added their voice by characterizing software engineering as a "*process of taking a specification and turning it into a software product*". This specification emphasizes the functional requirements of a system. The way the functional specification is developed follows practices that are used in fields such as industrial manufacturing. If these practices are followed in the formulation of a software specification, the process will become very mechanistic (Pressman, 2005:45). The end product, the piece of software will neglect the behavioural characteristics of organizations.

It should also be noted that the scope of SE (Schach, 2005) is very wide and includes the study of mathematics, computer science, sociology, philosophy, economics, management and psychology to mention but a few disciplines. However, all these various fields that inform the practice of software engineering, SE has largely been confined to the hard sciences discipline, in which each software product is viewed as implementing a system with a specific goal, having a definite boundary and is closed and deterministic. Developers are trained to assume that for any specific input to a software system, there should also be a related predetermined specific output. This definitely contradicts the behaviour of human activity systems. In addition, it is worrying that most software engineering practitioners are barely familiar with more than one of the disciplines mentioned above, especially the psycho-cognitive disciplines such as psychology. This alone is a disadvantage to the software development profession.

Current software engineering practices disregard the fact that the software product is not an end but is rather a means to an end: the implementation of organizational information systems. As a means to an end, during its development, the software product should take cognizance of the characteristics of organizational systems as supported by these three theories: the theory of organized activity (TOA), actor-network theory (ANT) and activity theory (AT). This also supports the argument that information systems have multiple goals and are non-deterministic and that their boundaries are not definitive but are permeable and open (Du Plooy, 2004). In fact, the discipline of software engineering should be regarded as a soft discipline in the fashioning of which Checkland's (1999) soft systems-thinking methodologies can play a major role.

It is important to see how the definition of software engineering has contributed to the way developers engage in software engineering practices. At the same time, the view given to organizations, their information systems and culture should be described in relation to the problems encountered in the development of software products. Organizations exhibit a dynamic attribute that cannot be represented by a static piece of software. As described above, there is a paradox in the practice of software development, with a dynamic environment being forced onto a static mode of representation. This paradox may safely be described as the software development problem.

4.2.1 The Software Development Problem

The software development problem was coined a '*crisis*' at the first NATO software engineering conference (Randell, n.d.). The issues discussed during this conference included project budgetary, quality and timing problems (Baker, 2006). As Brooks (1987) stated, the field of software development is characterized by projects that miss their schedules, exhaust their budgets before completion, and, finally, result in flawed software products. These issues are barely managed and satisfied during software development. Although many attempts have been done to address these problems, the software development problem still persists to this day.

In his contribution to the software development debate, Mullet (1999) cites two reasons for software development process continuing to be in a crisis. He attributes these to the lack of relevant software engineering (SE) education and the fact that SE is regarded as a craft. It is argued that many developers are concerned with the final product and not with the process itself. This problem is exacerbated by the separation of the software development and system development processes. Software developers should not separate the process of developing software from that of building organizational information systems, he added. Instead, the software development process should be regarded as integral part of the system development process.

According to Whitten *et al. (*2004), a system development process includes a set of activities, methods, best practices, deliverables and automated tools that developers use to develop and maintain information systems. Like the software development process a system development process, therefore, requires a system development methodology. In discussing system development methodology, Whitten *et al.* (2004) included the maintainability aspects of both the software products and the resultant information systems in the system development

process. Ensuring that the maintainability of the two is considered during development can also help to reduce the problems in both software and organizational information systems.

In another memoir, Basden (2001) decried the continual lack of return on investment from information systems investments. He tracked down these problems to software products and their characteristics which subsequently emerge in the final information system. He found that these problems originated during the developmental stages of these information systems. Pressman (2005) noted that the software product always carries the responsibility of delivering other products, such as information systems and their outputs. This piece of software, therefore, plays a dual role in its life cycle (Pressman, 2005). This dual role also puts an extra burden on developers, who have to ensure that the right product is developed before it is used in information systems.

Basden (2001) contended that there was something very wrong with the way in which the software product as a system artefact is developed. It is, therefore, very important to find and resolve the problems that lead to the development of flawed pieces of software. Another problem that has bedevilled the development of software products is that software developers have always construed the process of requirements analysis (from which the specification is built) as a process of eliciting plans and formal definition of procedures that should be enforced in an information system. This emphasis on formalization has resulted in neglect of the softer issues inherent in organizational information systems. In the end, the formal plans and procedures are expected to run the IS just as efficiently as the actual organizational system. It is pointed out that these formal plans often lack the business and domain aspects of the organizational system and that they only capture the formal system specifications that can be technically represented.

Furthermore, issues considered during the software development process play a vital role in shaping and determining the quality of the software product. It should be borne in mind that lack of quality is one of the issues that contributed to the software crises. Basden (2001) also noted four areas of concern that could be addressed to improve the quality of the software product. The first is on how the software artefact is fashioned for use. The second looks at the development of technology which we use to fashion the software artefact. The third focuses on how the software artefact is used and, finally, the focus is on the users' and developers' overall perspectives on the use of technology. This last aspect requires people to consider the social construction nature of the software product. Basden's concerns can only be addressed

by finding a software development methodology that integrates these four requirements into a software process model. These can be in addition to the focus on the design of reusable components and on the innovative elements of a software product design. These innovative elements of a software product represent the domain-related additions that make the difference between domain packages. These contribute to the life-responsiveness of the information system.

With regard to domain, De Oliviera *et al.* (2006) regard the lack of domain knowledge by software developers as the greatest problem in the development process. Although the processes of requirements elicitation and knowledge-gathering are very laborious, knowledge sharing and reuse of knowledge in software development is however, very limited. In these processes, one needs to explain the same concepts repetitively to different software development personnel. Developers have to study and learn the domain while simultaneously linking this to the tasks to which it relates. These tasks pertain to the problem domain that has to be addressed by the software product (De Oliviera *et al.*, 2006).

Brooks (1987) grouped the difficulties inherent in the software development process into three categories that is, the fundamental nature, the difficulties intrinsic in the nature of software and the accidents. He defines the fundamental nature of a software product as a formalized construct of interlocking concepts, sets of data, and associations among data items, algorithms, and invocations of roles. He noted that the explification of this essence or fundamental nature poses a big challenge in software development. The other two are much easier to indentify. The next section discusses some of the development practices that have been used to define the essence of a software product.

4.3 Software Product Development Practices

In this section, various practices being employed by practitioners in order to improve the quality of software products are discussed. As mentioned earlier, most methods are aimed at improving the production rate, quality and maintainability issues of the software products. Software kernels and software product lines are the two practices that will be discussed in this section.

A software kernel is a part of software system that is relatively stable over time and is used in a family of related products (Dittrich & Sestoft, 2005). The basic requirement of a kernel is to provide and encapsulate core functionality for the software product. It is the foundation upon

which other successive software products can be built or from which they can evolve. The purpose of a software kernel is to increase the reusability, production rate and, possibly, the consistency of production of software products. On the other hand, the software product line approach (SPLA) to software development can help developers to find the functionality, structure of the kernel, and also the structure that a kernel imposes on the applications based on it (Dittrich & Sestoft, 2005).

The Carnegie Mellon Software Engineering Institute (CMSEI, n.d.) defines a software product line (SPL) as a " set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."

The problem with the software product line approach is that, it is a predictive type of software reuse strategy and does not allow the time-dependent adaptation of the software product to the environment and context of the organization. This approach focuses on software artefacts that can only be packaged if there is a possibility that they may be used in one or more products that are in the same product line (Krueger, n.d.). SPL focuses on the creation of a portfolio of software artefacts that comprise a set of features shared in common. These are packaged together and stored. When reused, the software products can decrease the time to market, reduce costs and facilitate the evolution of software pieces.

However, the SPL development strategy is ineffective in providing flexibility, fast response times, capturing of semantics and context or in providing capabilities for mass customizations (CMSEI, n.d). SPL and software kernels have not had much success as development paradigms because they neither support nor encourage the development of adaptive systems. One of the major reasons for the lack of success of SPL and the kernel approach is that they do not allow a holistic transfer of analysis model characteristics, that is, the business, domain and specification models through to the design and, subsequently, to the implementation model. Because of this, they persist in neglecting the behavioural characteristics of organizations in their implementation strategies. The next section will discuss some issues of communication in software development.

4.3.1 Communication in Software Development

The failure of several software development processes or methodologies to transfer the full analysis model to lower levels of design and implementation can be attributed partly to poor

communication techniques in these process models. The communication is in two parts, namely, during requirements gathering and during the transfer of the requirements into the analysis model through to the implementation model. This section will consider both communication issues and, finally, highlights the major communication problems that are faced by software developers.

In all software development processes, the analysis phase is tasked with creating a true reflection of the environment of the organization. For a successful software development project, communication with users has to be given high priority in most of the developmental phases, including analysis. Harris and Weistroffer (2009), in particular, investigated several empirical studies and found a positive correlation between user involvement and system development success. They noted that users are more important during the preliminary phases of feasibility studies, requirements determination, and of the later stages of the design of input and output documents and the installation of the systems (Harris and Weistroffer, 2009). Lehtola *et al.* (2009) proposed the creation of solution concepts, a solution package that could be used to guide requirements engineering activities. Their proposal is based on the need to have a strong functional link between the business strategy and the software development process.

Garcia-Duque *et al.* (2009) further proposed another method for improving processes of gathering requirements, analysing them and later revising them. Their intention was to increase the fidelity to the business system requirements of the requirements gathered. They noted that this could be achieved through a continuous process of analysis, verification and revision. On the need to include the human context in software products, Fuentes-Fernandez and Gomez-Sanz (2009) provided a framework based on socio-psychological activity theory that could guide the gathering of social aspects of systems and, subsequently, include them in the software products. In addition, they saw a greater need for developing analytical tools that could be used to elicit the human context of business systems. All these endeavours aim at addressing the communication problem in software development. The communication problem is not limited to the analysis stage alone but extends to all phases of software development. The basic requirement is to address the collection of socio-technical systems requirements (Bryl *et al.*, 2009).

The analysis phase should produce a model that is descriptive and has the form of a computationally independent model (CIM) (Aβmann *et al.*, 2006). The CIM possesses as

little platform-dependent information as possible, at the same time ensuring that the customers' viewpoint is maintained. This customer can be regarded to as the business user. This computationally independent viewpoint captures the environment and requirements of the system. Many techniques have been used in traditional analysis modelling to ensure that this analysis model is expressed in terms of the problem domain. The intended relationship in communication that should exist between the analysis, design and the implementation models is illustrated in Figure 4.1.

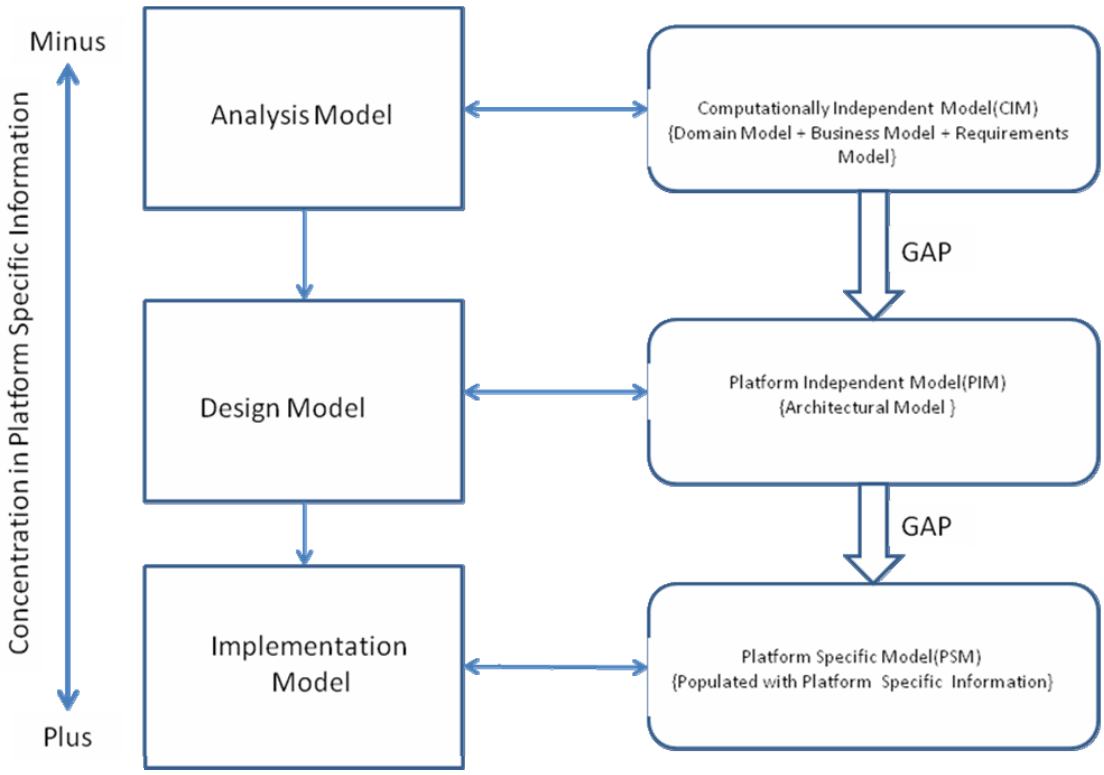

Figure 4.1: Communication Gap in Software Development

As discussed earlier, the analysis model is derived from the environment of the organizational system. This environment is characterized by the domain and business information, both of which are context related. In addition, the system requirements, which derive their fit from the organizational context, are also added. The purpose of the analysis model is to capture the triplet: domain model, business model and the requirements model. The requirements model manifests itself as the system specification. Also, the domain model is a product of the process of domain engineering. Bjørner (2008) lists understanding and capturing human behaviour as the prime purpose of this process. It should be noted that organizational human behaviour which comprise a bigger part of the domain model has to be described formally or informally and communicated throughout all the software developmental stages. This

domain model should capture the concepts used in the domain field as well as the relationships between these concepts. Bjørner (2008) also lists intrinsic, supporting technologies, management and organization, rules and regulations, scripts and human behaviour as the guiding principles for developing domain models.

The business model is tasked with capturing the organization's rules of business. Lastly, the requirements model, which models the system specification, is tasked with capturing the functional and non-functional system requirements (Aβmann *et al.*, 2006). On the other hand, the design model is the architectural model of the system and, at this stage; it should capture the system from the designer's viewpoint but should still be platform-independent. Lastly, the implementation model is gradually populated with platform-specific details as discussed by Aβmann *et al.* (2006). There is a gap between the different types of domain knowledge in the domain model and the substance and form of software artefacts that are constructed. To reduce this, Falbo *et al.* (2002) proposed an infrastructure specification that, with its semantics as captured by the domain model, could be used as input to the implementation phase of the software development process. Wand and Weber (1990) advocated a complete translation of the analytic model attributes through to the implementation model.

Software developers usually talk of formal system specifications. These generally refer to program code, thus implying that the specifications are platform-dependent. For specifications to be platform-independent, some informal specification methods should be found and used. Bjørner (2008) argues that this may require the specification to be drawn up in some sort of natural language. This has not been the case with all specifications, especially when developers arrive at the later stages of development. Bjørner (2008:62) decries the fact that "there are very many aspects of requirements that we today, 2008, do not know how to capture formally..." As a result, many analyses and design specifications do not represent the true world view of a system.

The insistence of developers on formalization results in the development of mechanistic development methods. Because of this formalization in these mechanistic development methods, from the analysis model through to the implementation model, the platform-specific information has been allowed to creep into the system. The major problem faced by software developers is that of translating all the characteristics of the analysis model (CIM), as shown in Figure 4.1, to the implementation model through the design model (PIM). This is normally because, at the end of the analysis stage, the system requirements are translated to a

specification model (SM). This SM is an instantiation of parts of the functions of the system. It focuses on those system aspects that can be formalized and the informal aspects are ignored.

As stated by Aβmann *et al.* (2006:257), "*a specification model is a prescriptive model, representing a set of artefacts by a set of concepts, their interrelations, and constraints under the closed world assumption*". The failure of the SM to transfer descriptive information captured by the analysis model to subsequent stages poses a serious problem in software development. In another attempt to improve communication in software development practice, Gunter *et al.* (2000) developed a reference model that could be used to bridge the gap between the analysis and design phases of software development and increase communication.

4. 3.2 Software Engineering Reference Model

The software engineering reference model developed by Gunter *et al.* (2000) consists of five basic components: domain knowledge, user requirements, software specification, software program and the programming platform. These are illustrated in Figure 4.2. This model is supported by Aβmann *et al's.* (2006) description of domain knowledge, in which they state that the domain knowledge, together with user requirements is used to describe the environment. On the other hand, the system to be developed is described by the program and the programming platform, the environment and the system being linked through the software specification. The software specification or model as portrayed here is a product of both the analysis model and the design model.

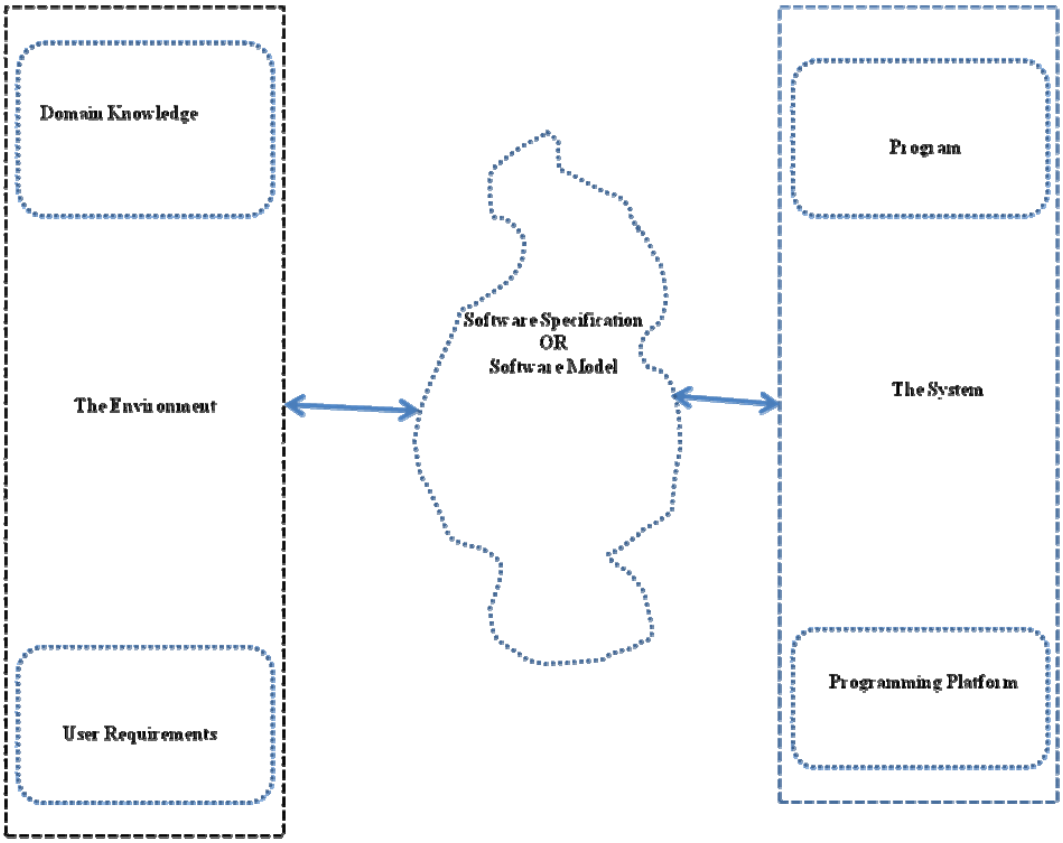

Figure 4.2: Software Reference Model *(Adapted from Gunter et al., 2000)*

This reference model (Gunter *et al.*, 2000) highlights the need for a good communication medium between the environment, the system, and the medium. In other words, there needs to be a mapping between the environment and the system, in this case, the system specification or software model that maintains the fidelity of system characteristics in the environment to the developed system. Gunter *et al.* (2000:54) concluded that a framework for "classifying and relating key software artefacts" is needed. The framework can be used to connect the formalization needed in artefact development to software engineering practice. In fact, Gunter *et al.* (2000:54) proposed a "systematic methodology for formal software engineering". This systematic methodology should be able to transfer all the possible system states in the environment to the system implemented. Although this reference model includes domain knowledge and user requirements in its discussion, it does not include the relationship between the analysis model as discussed in section 4.3.1 and the design and implementation models. At the end, it falls short of capturing the softer aspects of organizations in software products.

Another communication technique that has survived the test of time is the database conceptual schema. This schema has been used to mediate between the environment (user

interface), the applications (programs) and the storage facilities (databases) in almost all the systems developed to date. In the next section it will be argued that the way this schema concept has been implemented also greatly affects the way systems work in real life.

4.4 The Need for a Conceptual Schema

The conceptual schema was proposed in the 1970s by the American National Standards Institute (ANSI). The schema has been used for encoding knowledge in information systems (Sowa, n.d.). It captures and stores application knowledge of information systems (Sowa, 2000). The schema and the role it plays in information systems are illustrated in Figure 4.3 below.

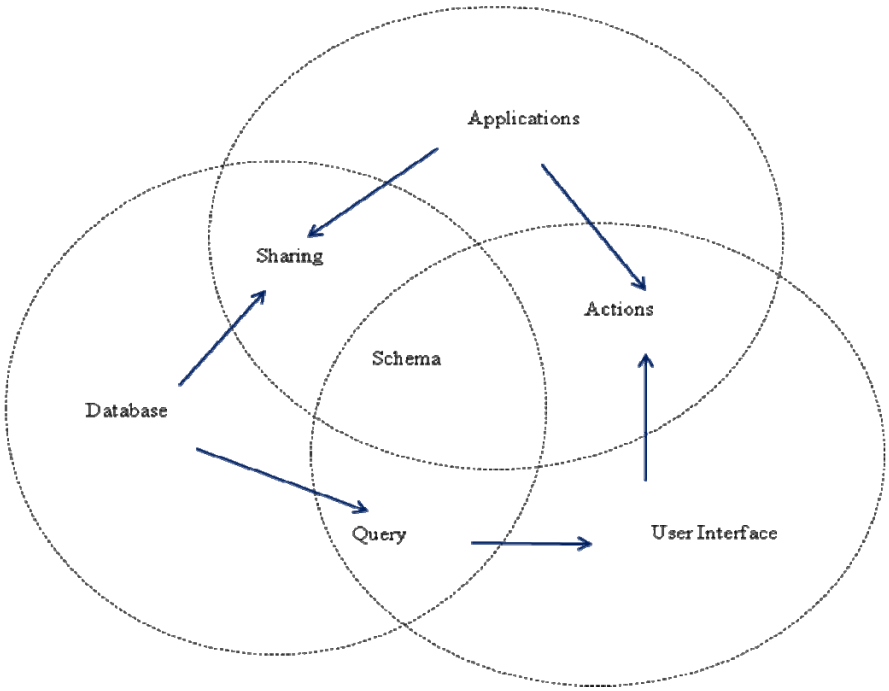

Figure 4.3: Components of an Integrated Information System *(Adapted from Sowa, 2000)*

As depicted in Figure 4.3, the user interface is the bridge between the system and users. The user interface is populated with facilities that allow it to access the database. By means of this user interface, users can query and edit contents of the database. At the same time, the interface facilities should be able to perform actions and provide services for the application programs.

Between the database and the applications, the database provides the data needed to run the application programs. This database allows data sharing and also provides a permanent

storage facility of data for the information system. Lastly, the application programs are the coded processes and activities that are required in any system to perform any prescribed task.

As the communication engine and the brains behind information systems (IS), the schema should provide common knowledge, application entities and relationships that exist in the organizational system. In a real world scenario, the duties performed or allocated to a schema are usually the responsibility of a human actor. Although human actors allow syntactic subscription of actions to perform any given task, they also use concepts in ill-defined contextual situations. In contrast, replacement of a human mediator by a schema that communicates at a syntactic logical level only reduces the requisite variety of the subsequent information system. Practically, the schema concept allows for an effective and efficient system design but is, however, very syntactic and is not adaptive. The major reason for the schema being syntactic is that it was developed and fashioned to interact with and fit to an already established database query language, the popular Structured Query Language (SQL). Sowa (1976) tried to circumvent the dictates of the schema by using conceptual graphs in querying and communication. However, this attempt has not yielded tangible benefits and until now, the system development fraternity is still subject to the mechanistic dictates of this schema. This section concentrated on some pertinent issues that remain unaddressed in software and system development. These issues continue to have a negative influence on the way in which software products are developed. Section 4.5 can be viewed as the epistemological and ontological grounding of organizational information systems and, hence, of the software products.

4.5 The Mechanistic Nature of Information Systems

"*There is a reason why computers have not yet become fervent natural language speakers. (It's not a matter of processing power and never will be): we simply are not programming them correctly.*"

El Baze 2005.

El Baze's (2005) comment points to the fact that, although developers can perfectly implement whatever they design, software products fail because it is the design itself that is grounded on improper assumptions. Software product designs do not take cognizance of the central role played by people in organizational information systems. With this in mind El Baze (2005) laments that "we simply are not programming them correctly".

In addition, the syntactic nature of software products is also the result of the philosophical assumptions that guide the development of software products. Embedded in these philosophies are the understandings given to a system, information system, systematicity and system formation (Gasche, 1986) as well as to the concept of system engineering, as discussed in Section 4.2 above.

Another reason is that current practices in information systems development are strongly based on a mechanistic world view. Based fundamentally on the functionalist paradigm, this view regards the world as being ordered, rational and unchanging. Monod (2007) strongly blames IS practitioners' unquestionable and faithful adoption and use of this rationality principle that make them believe that the world can be reduced to discrete functional units that can be represented as rules and algorithms. This world view also supposes that the knowledge and information that people have about the world can also be defined explicitly. In contrast to this notion, the knowledge and information used by people in organizations is mostly tacit and intuitive and cannot be identified explicitly and fed into rationalistic rules and algorithms. It is, therefore, important to motivate for a software development paradigm that rejects extreme rationalism and technological determinism as is currently the case and accepts that the world view is voluntarist, chaotic and subject to interpretation. This is the relativistic stance.

The use of the schema concept at the centre of the information systems (IS) to mediate between the user interface, database and applications has dictated the way in which software products are developed. Also most software development philosophies have been guided by the need for any system to accommodate the SQL as the query language, hence their strict adherence to the schema concept.

By definition, information systems consist of people, machines and processes that are required to accomplish specific tasks. As part of information systems, computing machines are clearly products of mechanistic thinking. Instead of limiting this mechanistic thinking philosophy to the development of machines, developers have also applied this mechanistic principle to the way people and processes work in organizations. This mechanistic way of thinking depends on the idea of a bureaucracy, of an explicitly formalized organization working only with explicit data. The mechanistic modelling of systems removes truth in the real world from the resultant model. This is called prescriptive modelling and can be

contrasted to descriptive modelling, in which the truth in reality has to be captured in reality (Aβmann *et al.,* 2006).

As stated earlier, the software product development process requires developers to exhaustively capture all the possible scenarios that can be assumed about a system. This, however, is difficult as it is not possible to capture and match the requirements of organizations that always have a running context that determines the next step of action in the software product. The failure to capture the running context results in the development of closed and predictable systems (Avison & Fitzgerald, 1995).

In addition to this, many organizational systems have databases that are populated with data objects that "ignore physical objects, processes, people and their intentions" (Sowa, 2000:56). It should be noted that computer systems do not have intentions of their own (Oinas-Kukkonen and Harjumaa, 2009). Rather, those who build and distribute the technology should inscribe the assumed intentions into the computer systems. These computers systems can be grouped into autogenous technologies (that shape the behaviour and attitude of those using them), endogenous technologies (technologies developed with user voluntariness toward attitude or behaviour change) and lastly, exogenous technologies (technologies that provide the means to personalize the assigned goals). The most important issue is that this role should be given to software developers and, most importantly, to the methodologies they use.

In addition, the databases pay very limited attention to the semantic content needed in information systems development. Its greatest disadvantage is that the technology that people have entrusted to run their information systems cannot deduce relationships between signs as can the human mind. In short, software development has followed an engineering, mechanistic approach that assumes that the tasks and steps to be performed by a software product can be predetermined and fully specified (Brown *et al.,* 2004). This is a rational approach that is in line with developers' rational view of the world. These rationalists believe in the "power of good representations to predict and control social change" (Hirschheim *et al.*, 1995:3). To improve the usefulness of the software systems, developers need to combine and evenly balance a mechanistic understanding of computing machines with a romantic appreciation of the complexity of people, social organizations and information use. There must be a balance between the structural and functional views given to information systems.

This brief background to the mechanistic nature of information systems will lead us to the discussion of specific philosophies that are a foundation of these aspects.

4.5.1 Mechanistic Systems, Systematicity and System-Formation

According to the Theory of Organised Complexity (TOC) (Checkland, 1999:78), systems in general exhibit a general hierarchy of levels in which each level is more complex than the level below it. Each such higher level also has emergent properties that are not found at lower levels. It should be noted that these emergent properties are the result of system formation, in which the whole exhibits characteristics that cannot be found in the individual sub-systems that combine to form it. As noted by Checkland (1999:78), *"neither a one level epistemology nor a one level ontology is possible"* to describe the sum total of the subsystems. This expresses the concept that, in a hierarchy of systems forming the whole, each level has different distinct epistemological and ontological views. In other words, the views of the lower levels of the system can never be the same as those of the whole system. Put in another way, the behavioural characteristics of a subsystem, when added together will generate a system whole that has behavioural characteristics completely different from those of its constituents. The interaction of the components creates some emergent properties that are a by-product of the interaction and these manifest themselves in the whole as new characteristics. This principle supports the notion that an aggregation of mechanistic components cannot have the same properties as the whole. This also brings in the two notions of systematicity and system formation. In other words, systematicity that led to the mechanistic development of systems looks at the extent to which a system can be regarded as an ordered, hierarchical arrangement of components. Its focus is the machine and the artificial versus the preservation of the natural that is encompassed in the romantic world view. System formation in turn looks at the ordered, organised building up of the whole system from its components.

In explaining these two terms, it is important to define the concept '*whole*' as a synthesis or unity of parts. These parts are so close that their activities and interactions are affected by the closeness of these parts as a result of the synthesis. Information systems can be taken as a subset of all the possible states that make up the whole system. Such a view posits information system infrastructures as artefacts that are not discrete pieces of components or fragmentary constructions. Instead information systems should be conceived as "artefacts made up of a continuous connection; chained arrangement of its constitutive parts. These

constitutive parts, if fragmented and reassembled, one cannot come up with the original whole" (Gasche 1986:5-7) *(sic)*. This contrasts with Aristotle's maxim that the whole is equal to the sum of its constitutive parts. When the whole is broken up, it loses its requisite variety by negating some other possible states. At the same time, when the system is reassembled from the parts, the theory of organized complexity explains the introduction of emergent properties that never existed in the original whole. This now constitutes a paradox, where one cannot holistically get an organizational whole from the products of its deconstruction, that is, the constitutive parts.

In its totality, a general system that is reconstituted (developed) from the mechanistic principles of reductionism cannot have that unity of purpose or focus and/or the horizon of meaning, sense and context that gives it the attributes of a total or a whole system (Gasche 1986).

Although, through the application of reductionism, systematicity and system-formation have been used to construct general systems (Gasche 1986:8), the system components could not be reunited into a "one, well rounded-off system". Gasche (1986) sums it up by asserting that, although reductionism is a very good condition of successful idealization of information systems development, the resultant information system itself lacks idealization (Gasche 1986). In relation to the principles of systematicity and system formation, there is also another principle, that of romanticism.

4.5.1.1 Romanticism

Romanticism in systems can be described using Gasche's (1986) notion of anti-systematic thought. This is related to the anti-positivist notion described in Chapter 2. Romanticism argues for the negation of systematicity and system-formation while, at the same time, allowing the concept of "the fragmentary …" and the process of deconstruction of the greatest totality to be based on the former. According to Gasche (1986:7), deconstruction concerns itself with finding the limits of the acceptance of "systematicity and system formation". Deconstruction is a conceptualization process of what constitutes the so-called "general system". As Gasche (1986:7-8) noted: "the general system is not the universal essence of systematicity; rather it represents the ordered cluster of traits of possibilities which, in one and the same movement, constitute and deconstitute systems."

The paradox that faces system developers emanates from the fact that one cannot tackle a system development project without breaking it up into manageable chunks. On the other hand, the whole cannot be reconstituted from these chunks. A development approach that reduces the gap between these two poles, one that is able to reintroduce the romanticism that existed in the original system in the developed systems, has thus to be found. This section supports the need for a software development methodology that allows the deconstruction of original systems to the constitutive parts and then allows the reconstruction without losing the softer human aspects.

By assuming the romantic world view, IS developers consider a holistic view and an acceptance of the organizational system, where culture and social context play a part in the execution of a task. In the eyes of romanticists, "processes and change" are at the forefront of system "contemplation, understanding, interpretation and feeling" in contrast to the structures and systems of the mechanistic world view (Dahlbom and Mathiassen, 1997:501). In this view, change in organizational systems is taken as "unpredictable and beyond human control, the expression of hidden and unknowable forces" (Hirschheim *et al.,* 1995:3).

During system development, developers need to consider the structural features of systems that should be conceived in any development process if the system is to achieve its goal. As noted by Wyssusek (2004:4303), at this juncture, information systems analysis and design "should not solely be guided by an instrumental-technical and / or a practical-hermeneutical cognitive interest but also an emancipatory interest…" In this case, developers can, therefore, overcome their long "self inflicted cognitive constraints" (Wyssusek, 2004:4303). Many information systems analysis and design techniques take the ontological position that neglects the modelling of softer human issues in organizations. The process of structurally breaking up the system features during analysis and design, and later recombining them during implementation so as to recreate the whole, poses serious limitations to the current reductionist processes in many system development approaches.

4.5.2 Transition to Romantic Systems

Problem-solving in the interpretive and neo-humanist paradigms cannot be guided by syntactic rules and formalizations as in the functionalist paradigm. On that note, IS developers are urged not to concentrate on tractability and objectivity in this field. Any emphasis on these two aspects distances and isolates the resultant artefact from complexities of everyday social changes. This problem is prevalent during the analysis phase, during

which requirements are gathered, and is aggravated during the transition from analysis to design of the systems. At the implementation stage, the people aspect of organizational IS is almost completely forgotten.

Yu (1997) noted that many requirements-modelling techniques focus on the completeness, consistency and also the automation of system requirements verification: neglecting the early requirements tasks that are stakeholder oriented. These are tasked with checking "how the intended system would meet original goals" (Yu, 1997:1), including, but not limited to, why and how the system will address stakeholder needs. It is more important to leverage the requirements engineering effort on the reasons why requirements are needed and gathered than on the specification of what the system must do (Yu, 1997). In fact, as Roque *et al.* (2003) put it, there is more to *requirements than elicitation.* Requirements are an everyday social construction, in which human and non-human actors participate. This position requires information system (IS) developers to look holistically at the organizational context. The organizational context includes information technology (IT), the people and procedures in the organization and the interaction of these three actors. The mechanistic nature of systems running in organizations is the result of looking at only a single part of the system, the *serving system*. The serving system consists of the IT infrastructure only. The *served system*, which consists of people in the organization, is generally neglected (Kawalek & Leonard, 1996). The constituents of the served system are lost during the requirements gathering stage, and the few that are captured during this stage are also lost during the transition to the design stage.

Current system development methodologies typically force many organizational systems to embed their business rules, organizational culture, practice and their human aspect in the technological side of IS. It is, therefore, necessary to motivate for developmental methods that liberate the human aspects of the organization from the bondage of technology. Kawalek and Leonard (1996) motivate for a winning configuration in which the human and technological aspects are evenly balanced.

In many systems, current modelling techniques, in which systematicity and system formation are profound, the task approach described earlier is used. As Keen (1988) noted this approach encourages an IS culture of thin technical orientation. He then argued for a role-based approach in which the IS and user relationship is evenly balanced. The task approach works efficiently in an environment in which discrete processes can be used. However, the IS

environment is not that discrete. In support of Keen (1988), Lehman (1994) described a domain that is served by software applications as continuous, irregular and dynamic. In contrast, the software products that serve this domain are themselves distinct, bounded and, unless human decision and action has changed them, are static. Software products have long been used as the technological artefacts that map the social organizational system to information systems. It is, therefore, necessary to focus any attempt to reduce the mechanistic nature of information systems on the software product. The software product is derived from the software model (Gunter *et al.,* 2000) as a software specification; hence the software model becomes the blueprint for the software product.

The software model as a piece of organizational representation and also as the blueprint for the final software product is viewed by Lehman (1994) as an estimate, essentially incomplete artefact with entrenched assumptions. This should not be surprising since, as already discussed, the software model is a product of the mechanistic development paradigm. More often than not, it lacks the domain and business model aspects of organizations. The next section looks at three theoretical frameworks that can be used to appraise organizational systems.

4.6 Activity Theory, Actor-Network Theory and Theory of Organized Action

This section uses the three philosophical theoretical frameworks: Activity Theory (AT), Actor-Network Theory (ANT) and Theory of Organized Action (TOA), to look at organizational systems. The problems caused by the reductionist tendencies of systematicity can be viewed through the principle of human activity theory as a lens. Human activity theory is derived from the principles of activity theory.

4.6.1 Activity Theory

AT framework has been used for a very long time now to explain and understand human activity systems. An activity is described as the smallest indivisible, action-oriented and goal-directed process that can be found in a system. Taking an activity as a basic unit of analysis in organizations, we use AT to explain the "coherence of individual actions in a larger social context" (Roque *et al.*, 2003:112). An information system (IS) is considered as an assembly of individual activities that work in synergy to accomplish an organizational goal. These activities are the ones upon which the task approach of IS development are based.

As activities make up the IS, each activity in turn is composed of actors. An activity can be described as an organized assembly of actors that work together to accomplish a stated goal. The activity is then "driven by peoples' needs" (Fuentes-Fernandez and Gomez-Sanz, 2009:3) and as a process, it accepts objects as input and the output should be able to satisfy peoples' needs. An aggregation of the activities, therefore, constitutes an information system. The actors in an activity can be humans, technological artefacts (hardware and software) or a combination thereof.

These basic components of information systems have to communicate through a mediator in order to satisfy organizational requirements. The mediators are also responsible for acting as communication channels between the actors. Although they are actors, software products are also mediators in the system. They, therefore, mediate between individual humans, between humans and technological artefacts (hardware and software) and between purely technological artefacts. This state of affairs, in which technology and people coexist, leads to the understanding that mediation software products work in a socio-technical human activity system.

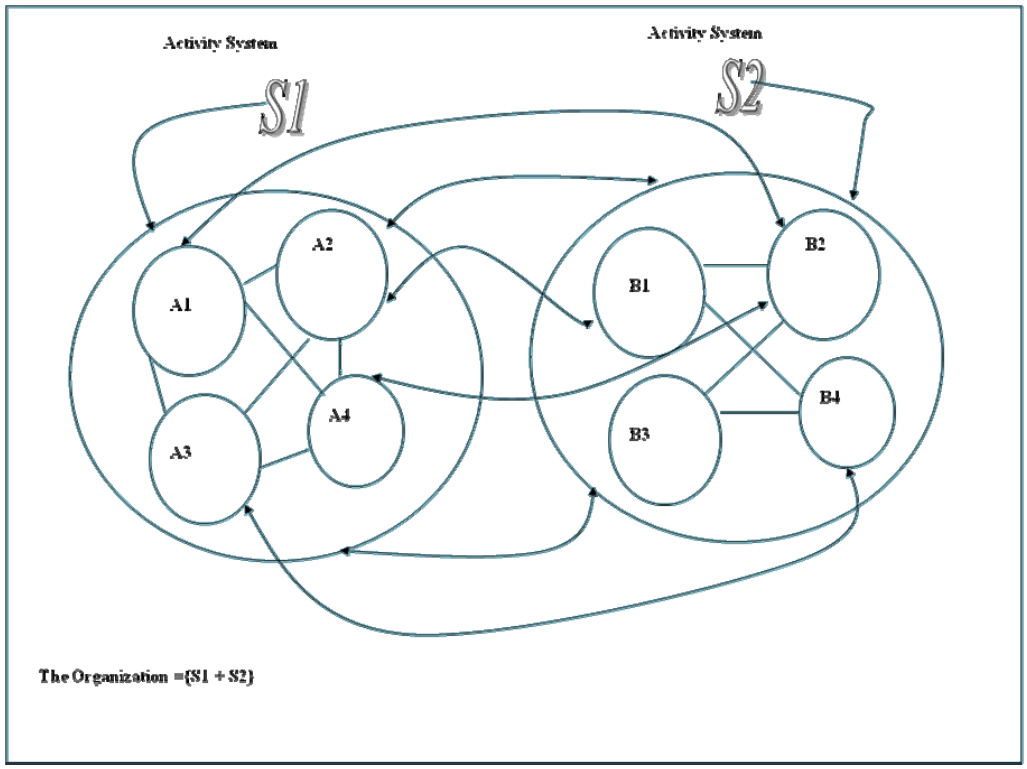

Figure 4.4: The Concept of an Activity System (*Adapted from Roque et al., 2003)*

Figure 4.4 shows A1 - A4 and B1 - B4 as individual activities and activity system S1-{A1 - A4} and activity system S2-{B1 - B4} are different activity systems. Besides being activities, A1 - A4 and B1 - B4 are also referred to as actors. S1 + S2 give rise to yet another combined activity system: the organizational system.

In practice, the goals for activities A1 - A4 and B1 - B4 can be determined *a priori* but it is difficult to determine the goals which result from their interaction to make activity systems S1 and S2 respectively. This can further be explained using Checkland (1999)'s theory of organized complexity described in Section 4.5.1 above. In addition, the combination of S1 and S2 to make "O", the whole organizational system, becomes even more complicated. This section will be used to appraise organizational information systems using activity theory as a lens. The human activity theory can be used to explain the non-deterministic and complex behaviour of information systems.

As discussed earlier, the current software development methods have been designed to follow the Aristotelian principle that the sum of the parts is equal to the whole. Owing to the inclusion of people as actors in these systems, these methods have failed to consider the human behavioural aspect of activity systems. In the behavioural sciences, the whole may not necessarily be equal to the sum of its parts. On that note, determination of the nature of a process (activity) and its outcomes and "ignoring changes in motives and goals, ignoring actors, human and non-human, ignoring the multiplicity of disciplinary agencies involved" have been the reason why information systems often fail (Roque *et al.*, 2003:113). The software product as the mediator should be fashioned so as to accommodate all these contingencies. AT can, at least, be used to explain the gap that exists between organizational IS and IS products.

4.6.2 The Social Network Aspect of Activity Systems

As already discussed and with reference to Figure 4.4, each activity system A1 - A4, B1 - B4, S1 and S2 can be referred to as actors. Within each activity, a software product mediates and creates communication channels. Since each activity system consists of a network of interacting actors, each activity system may, therefore, be referred to as an actor-network. In this regard, Latour's (1999) Actor-Network Theory (ANT) can be used here as a theoretical lens to explain the social interactions and relationships that exist between each actor in the activity system.

ANT regards each component of an activity system as an actor that is influenced by other actors through the mediation of these actors. The actor, as an "author of inscriptions", is a network itself and centre of translations. Each actor is also influenced by the relationships established by itself as a node in the network in which other actors participate (Roque *et al.*, 2003:113).

Because of the presence of humans in IS, the actors participating in the IS as an activity system exhibit both human voluntarism and technological determinism whose interplay results in the emergence of complex social characteristics. In the spirit of ANT, any actor loaded with either inscriptions or translations, whether a mediator or the mediated, should be fashioned in such a way that it is allowed to evolve and co-evolve with other actors. It should be allowed to adapt to the ever-changing requirements of the environment. The environment in which these processes take place is not static but dynamic, continuously changing and adapting to new demands. The question, therefore, remains as to how to make the software product capable of accommodating these responsibilities.

4.6.3 Human Activity Systems (HAS)

Human beings have never been predictable. Their behaviour is always changing. Suchman (1987) contended that it is very difficult, if not impossible, to consider all contingencies in an *a priori* prescription of a human action. Rationalization of human actions *a priori* or *a posteriori* overlooks much detail that is situated in the running context, such as the detail taken during a course of action in everyday life (Roque *et al.,* 2003).

Roque *et al. (*2003) urge developers to guard against neglecting variability and independence during the development of IS products. They add that reliance on procedural and functional descriptions of organizations and individual roles as complete accounts of the social dynamics in an organization should be discouraged (Roque *et al.,* 2003). In mapping organizational dynamics, software development should not neglect the contextual settings of the system. Contextual information is, however, tacit and is shared among actors in the same organization. Software developers have always neglected the contextual characteristics of information systems (IS) because of their tacit attributes. Up to now little research has been done to incorporate context and intuition into the development of information systems (Beynon *et al.,* 2008). Another theory, the theory of organized activity (TOA) which is related to activity system theory, is discussed in Section 4.6.4 below.

4.6.4 Theory of Organized Activity (TOA).

Like the human activity system (HAS), Holt (1997) developed the theory of organized activity (TOA) that is also based on human (organized) activities. An organized activity is a dependent variable of the social interaction of people in a particular setting. Looking at an IS as an activity system, Cordeiro and Filipe (n.d.) view the technical aspect of IS as playing a supporting role to the organizational human activity.

They also describe the human action, that is, the action performed by a human actor in this actor-network as comprising interests and actors. These interests, together with the actors, are, therefore, responsible for the actions. While humans can have interests, technical machines cannot have interests and, therefore, cannot be assigned any organizational responsibilities. The end result is that the technical aspects of IS cannot perform actions (Cordeiro & Filipe, n.d.).

In short, although technological artefacts may be components of an organized activity, their failure to inscribe interests places them at a disadvantage if they are to be assigned any responsibilities for an action. The development of a software product as a technical artefact should, therefore, include ways that allow them to be assigned responsibilities. We can refer to this notion as humanizing the technical artefact, in this case the software product. The software development approaches that are employed in the development of the software product should be revisited and adjusted so as to allow the incorporation of methods that will capture the human and social nature of the activity systems.

These three theories, AT, ANT and TOA, discuss and support a very important characteristic of organizations and their information systems. In reflecting on software development, these theories urge researchers and developers of IS products to consider organizational culture, the context in which the software products will be used, how to capture the tacit information in organizations, as well as the non-determinism of socio-technical construction systems such as organizations. As discussed earlier, these three theories can be used as a basis for a conceptual grounding of romantic information systems. A conceptual framework for romantic information systems can be described as:

> *a framework that allows the development of socially-constructed systems that capture and maintain the softer elements of organizational systems such as culture, social context, and*

semantics and to a certain extent pragmatics. These systems must be adaptive, dynamic, evolvable and innovative.

The whole idea is to bring some intuition, tacit information and meaning into the software product. This characterization may seem farfetched, but, existing literature like Weber (2003), Hohmann (2007), Yu (1995, 1997), Beynon *et al* (2008) and Soffer *et al*. (2001), to mention but a few, are already calling for a need of a romantic framework like the one proposed in this study.

Having motivated for a new way of developing software products in previous sections, in the next section the researcher looks at the approach to, or the methodology requirements of, a software development process.

4.7 The Approach versus Methodology Debate

The purpose of this study is to motivate for a new software development approach. As such, this section delves into the fundamental tenets that are considered as differentiating between an approach to and a methodology for software development. Starting with an approach, Checkland (1999) refers to an approach as a way of going about tackling a problem while Iivari *et al*. (1998:166) regard it as a "class of methodologies which share the fundamental concepts and principles for ISD". Such an approach may not, however, be very prescriptive regarding the method to be used when tackling a problem and may not be required to provide methodology instances. This implies that, an approach may be formulated without any accompanying methodologies (Iivari *et al.,* 1998).

Brown *et al.* (2004) consider an approach as the lighter form of a process. They add that, unlike methods, an approach is less prescriptive than a method and that adapting it to specific scenarios is fairly simple. Furthermore, an approach, by its very nature, can be developed into one or more specific methodologies. In the field of software development, Roque *et al.* (2003) characterize software development (SD) approaches as classes of methods that map areas of similar methodological thinking. Approaches look at methods that share goals, guiding principles, fundamental concepts and principles for the SD process. These approaches are derived from a specific paradigm, for example, structured, object-oriented and agile approaches. Software development approaches or methodologies are developed by method engineers (Gonzalez-Perez & Henderson-Sellers, 2006). In developing these approaches method engineers use concepts in the method domain, that later need to be "instantiated by software developers when they apply the methodology" or the approach.

In software development, methodology is defined by Schach (2005) as the science of methods. Pressman (2005) further characterizes a software development methodology as a framework that can be used to structure, plan, and control the process of developing software products. He goes on to state that the methodological framework consists of the two elements below:

- A software development philosophy: this is the approach to the software development process; and
- Some tools, models and methods that can assist in the development of software process.

Pressman's definition is likely to mean the same thing as Roque *et al.* (2003) and Iivari *et al.'s* (1998) characterization of an approach. Hence it shall be regarded as such. A methodology may also be regarded as a formalized approach or as a combination of steps and deliverables that are needed for the development of a software product (Dennis *et al.,* 2001). This, however, takes place at a lower level of the hierarchy than the approach discussed above. In contrast to an approach, although a methodology also includes the philosophical underpinning found in an approach, Benson and Standing (2005:203) also list "phases, procedures, rules, techniques, tools, documentation management and training for developers" as components of a methodology.

Characteristically different, but somehow similar, Bjørner (2008:41) defines a methodology as a "study of and knowledge about methods" which according to Gonzalez-Perez and Henderson-Sellers (2006) comprises of methods, techniques and tools. To avoid confusing methodology with method, Bjørner (2008:41) goes on to describe a method as "a set of principles for selecting amongst, and applying, a set of designated techniques and tools that allow analysis and construction of artefacts". In this he supports Benson and Standing (2005) but also emphasizes the prescriptiveness of methods, techniques and tools that are used in a software development methodology.

In turn, Bjørner (2008:41) describes a principle as "an accepted or professed rule of action or conduct…" He also regards a technique as a "specific procedure, routine or approach that characterizes a technical skill" and views a tool as "an instrument for performing mechanical operations, a person used by another for his own ends… to work or shape with a tool" Bjørner (2008:41).

A term that requires further elucidation is the word “paradigm”. Schach (2005:25) characterizes a paradigm as a style of software development. This is a way of doing something as contrasted to the actual meaning of “a model or pattern”. Of importance is the fact that paradigms are not disjoint but may overlap in terms of approaches and methodologies. It should be accepted that different researchers and authors have different definitions for approach and methodology, including the methods. This is further reflected in Table 4.1 below which contains a list of some of these definitions from different authors.

These excerpts are however, not exhaustive. There is much literature that discusses these three subjects in different expert disciplines and, therefore, readers are urged to acknowledge the varied contextual interpretations given to them. The definitions in Table 4.1 are, however, derived from authors and researchers in the field of software development and information systems. This makes them relevant to the discussion in this chapter.

Table 4.1: Characterizing Approach, Methodology and Method

Characterizing Approach, Methodology and Method	
Concept	**Concept Characterization**
Approach	An approach is a set of goals, guiding principles, fundamental concepts, and principles for the system development process that drives interpretations and actions in system development (Iivari *et al.*, 1998).
Methodology	A methodology is defined as an explicit way of structuring one's own thinking and action. It contains models and reflects a particular perspective of reality based on different philosophical paradigms. A methodology implies a time-dependent sequence of thinking and action stages (Jayaratna, 1990).
Methodology	A methodology is a set of recommended steps, approaches, rules, processes, documents, control procedures, methods, techniques and tools for developers, which covers a whole life cycle of an information system (Repa, 2004).
Methodology	A methodology is a collection of procedures, techniques, tools and documentation aids which will help the systems developers in their efforts to implement a new information system. A methodology will consist of phases, themselves consisting of sub phases, which will guide the systems developers in their choice of the techniques that might be appropriate at each stage of the project and also help them plan, manage, control and evaluate information systems projects (Avison & Fitzgerald, 2006).
Methodology	A methodology can be interpreted as an organized collection of concepts, methods, beliefs, values and normative principles supported by material resources (Hirschheim *et al*., 1995).
Method	A method is regarded as a path or a procedure by which the developer proceeds from a problem of a certain class to a solution of a certain class. The steps of a method impose some ordering on the decisions to be taken during development (Jackson, 1981).
Method	A method defines what should be done in a particular phase of the Information System (IS) development process. A method is always based on a particular approach (functional, data, object-oriented). A method always covers just part(s) of the IS development process or some point of view such as data, function, hardware (Repa, 2004).

The discussion in Table 4.1 positions an approach at a higher level of abstraction, descriptiveness and flexibility than does the methodology. The most important distinction between these two is the fact that methodology is more prescriptive, that the steps to be followed may be too limiting and are clearly defined. In an approach, a myriad of methodologies can be formulated. The most important aspect of a methodology as discussed by Jayaratma (1990) is the fact that the philosophical thinking in a methodology is a time-dependent sequence of activities which also accommodate some thinking, principles and normative principles (Hirschheim *et al.,* 1995). In light of that, Truex *et al.* (2000) warn methodology engineers to realise that most of the software development activities may not fit in the methodological frame that they are accustomed to, especially at very low process levels. This a-methodological frame may necessitate the omission of these activities in a

methodology layout. This is despite the fact that they encourage methodology engineers to clearly state the notation and guidelines to the use of their methodologies. A closer look at Table 4.1 shows that a method is derived from a methodology, in other words the degree of prescriptiveness increases as well.

Referring back to approaches, there are several ways of categorizing these. Some may use the dichotomy between hard and soft systems approaches and some may use the paradigmatic distinction encompassed in the methodology such as structured, object-oriented and agile approaches. In this research a paradigmatic classification according to how one perceives the world view has been used to categorize them under the realistic or relativistic paradigms. (*See Section 6.4*.) Using this classification, three categories are found, that is, the traditional or structured, approaches in transition and behavioural approaches. (*See Figure 6.1*.) The approaches using the methodological process followed during software development are discussed in Section 4.8 below.

4.8 Software Development Approaches

According to Roque *et al.* (2003), software development (SD) approaches lie between software development (SD) paradigms and SD methods. However, methodology, approach and paradigm apply to the whole process of software development. Currently, three SD approaches are found: the structured approach, the object-oriented (OO) approach and the agile approach. The discussion below focuses on the basic tenets that these approaches address. These tenets focus on the philosophical thinking, that is, on the beliefs and values that are enshrined in the approach.

4.8.1 Structured Development Approach

The structured development approach can be divided into process-oriented and data-centred approaches. Techniques such as the functional decomposition diagrams (FDD), data flow diagrams (DFD), entity relationship diagrams (ERD) and data structure diagrams (DSD) have been employed to model processes and data in this approach.

The structured approach is a reductionist type of approach which assumes that a system can be broken down systematically into a hierarchy of functions (processes) and sub-functions (sub-processes). This approach also uses deductive reasoning (moving from the general to the specific) (Brown *et al.*, 2004). The hierarchy theory discussed by Checkland (1999) can be used to explain the philosophical grounding of the structured development approach. The

hierarchy theory stipulates that systems can be regarded as having different levels of complexity. This difference in complexity is noted between levels either at a higher or lower aggregation level of the system. In short, it accepts that systems can be broken down into simpler subsystems and that the whole can be formed from an aggregation of its parts. In between levels, however, there are fundamental differences in their structure, complexity and composition.

The structured development approach is well grounded in the principles of systematicity and system formation, as explained in Section 4.5.1. It concentrates on analysis and algorithms but pays very little attention to synthesis or to relationships between parts during the process of software development. Several software-development methodologies based on the classical waterfall process model and some modifications have been used within the confines of this structured approach. The next section will discuss the agile approaches.

4.8.2 Object-Oriented Approach (OOA)

The item of consideration in the object-oriented approach (OOA) is the object. The object is designed in such a way as to encapsulate both data and procedures, unlike structured approaches which treat these two separately. System behaviour is an attempt to look at synthesis and the relationships depend on the interaction between the objects of the system. Objects sharing common characteristics form a class and all the different classes found in a domain area form the system. This contrasts with the hierarchical decomposition of functions found in structured approaches. Relationships between classes are considered and this adds to the semantic richness of the system to be developed (Brown *et al.*, 2004). However, the fundamental philosophical groundings in OOA are not very different from those of the structured approaches. Paradigmatically, both rely on the functionalist paradigm and the realist stance.

4.8.3 Agile Approaches

Agile approaches borrow their philosophical grounding from the theory of adaptive systems (Highsmith, 2004). Systems should be agile and adaptable to the environment in which they operate. A host of methods summarized in Brown *et al.* (2004) have been used in this line of thinking. The major differences between agile approaches and the traditional (structured and OO approaches) that are important to the development of information systems are summarized in Table 4.2 below.

Table 4.2: Philosophical Groundings of Software Development Approaches *(Adapted from Brown et al., 2004: 4139)*

Philosophical Groundings of Software Development Approaches		
	Traditional Approaches	**Agile Approaches**
Fundamental Philosophical Assumptions	Project proceeds in linear fashion. Deterministic approach that eliminates uncertainty through reasoning.	Feature-driven and proceeds in an iterative, evolutionary manner. Ambiguity and uncertainty are solved through cycles of rapid feedback and continuous improvement.
Process Model	Waterfall, spiral, etc. They emphasize linear sequence of process steps.	Evolutionary development model proposed by Gilb (1985).
Systems Thinking	Hard systems.	Hard and soft systems.
Thought Process	Driven by strict, predetermined rules established by the process model, which basically makes one react in predictable ways to unusual situations that might arise.	Urges developers to use patterns to solve problems by relying on their ability to innovate, depending on the contingency.
Dealing with Complexity	Assumes that complexity and ambiguity can be predicted, measured and corrected.	Deals with complexity and ambiguity by using the ingenuity of people and relies on rapid feedback and adaptability.
Customer Involvement	Not directly involved in the development process.	Mandatory, active participation throughout the development process.
Team Composition	Relatively homogenous.	Self-organizing teams involving relevant stakeholders who may have diverse perspectives and disparate goals.

It is worth noting that Brown *et al.* (2004) classified agile approaches as operating in the neo-humanist paradigm. In practice, however, these approaches seem to operate in both the functionalist and humanist paradigms. This is supported in Table 6.3 by the many incidences of techniques that are used in both paradigms, as well as by its classification under the paradigms in transition. (S*ee Section 6.4.*). Although some of the agile techniques fall within the humanist paradigm, the problem they face is that, after capturing the human attributes

inherent in organizational systems, they lack a process for including and maintaining these characteristics in the software products.

It can be seen that these current software development approaches place a greater premium on processes and technologies than on people and their behaviour, a notion that is strictly grounded in hard systems development methodologies. Roque *et al.* (2003), as proponents of a new approach to software products development, proposed some goals that should be taken into consideration if the software products are to capture the romanticism inherent in organizational natural settings. These goals have to be grounded within a certain philosophical underpinning for them to satisfy the development approach.

4.8.4 Goals of a Software Development Approach

It is accepted that each software-development process requires guiding principles, methods, tasks and activities that are guided by a specific body of knowledge. By the same norm, the software development approach in turn dictates the structure and form of the software development process to be followed. A software development process, life cycle or software process can be conceived as a structure imposed on the development of a software product.

Accordingly, each development approach should at least satisfy the following goals in addition to many other paradigmatic requirements. As discussed by Roque *et al.* (2003), some of the goals mentioned here require an approach that would:

- Allow software developers to frame the SD activities, supported by the relation between context and mediators of the activities that mould that context;
- Achieve an understanding of the SD activities on the proposed framework, viewing IS development as a social-technical phenomenon within cultural and historical envelope;
- Deal explicitly with contextuality in SD as the key to performing emancipatory movements; and
- Capture semantics, pragmatics and context in the software product.

As emancipators, software developers are able to anticipate users' needs and are thus able to produce artefacts that have the capability to capture third-party understandings of a situation.

These goals for software development approaches led to the discussion of requirements for developing romantic software products.

4.9 Model Requirements for a Romantic Software Product

This discussion enforces some requirements to any approach that is intended to yield romantic software products. The requirements are derived as a summary of the existing literature in the field of software development that has been discussed in this chapter. These requirements include but are not limited to the following:

- The software model should be sufficiently broad and flexible to accommodate a variety of possibilities.
- The software model should be easily actionable but should also allow loose coupling between applications and data sources.
- The software model should capture the context of the situation. This will ensure the capturing of all possible life states of the system.
- The software model should be sufficiently broad (abstract) to allow it to adapt to various and ever-changing sets of problems. It should allow for a dynamic transition between possible states of the system depending on the context.
- The software model should also be deep enough to provide a rich, detailed and valuable context.

Since the software model is tasked with the transcription of possible system states to the software product, it should have some characteristics that meet these requirements. With this in mind, the development of romantic software products should be guided by a framework that possesses sufficient obligatory passage and translation points (Introna, 1997; Mavetera, 2004b). These obligatory passage and translation points will allow the mediation of different contexts for different organizational software products. The issues raised in this chapter that can guide work on the development of romantic software products are summarized in the next section. In line with these issues, a software development approach that will ensure the development of romantic software products should encompass all or most of these requirements.

4.10 Software Development Issues

Reflecting on the discussion of this chapter, we discover that, from the beginning, software development approaches have been grounded on philosophical frameworks that use the realist world view as their cornerstone. This world view, in conjunction with the functionalist paradigm, views the served system separately from the serving system. Throughout this chapter, a consistent argument has been followed that motivates for a shift from the mechanistic software product-development approaches currently in use to one that allows the development of romantic software products. This approach is grounded in the relativistic world view and the interpretive neo-humanist paradigm. In moving to this relativistic stance, the following issues have to be considered:

- There should be a switch from hard systems approaches to soft system approaches.
- The methodological approach should have a way of capturing the dynamic nature of the ever-running context in organizations. On this note, the approach should ensure that software developers are able to study organizational environment and live in it so that they can have a situated practice and experience this practice before they embark on any software development project.
- The approach should ensure a transition from a task-based approach to the role-based approach.
- Developers should be able to build a language community with all stakeholders, that is, there should be a linguistic model that could be used to negotiate a shared understanding of the concepts found in a system. This requirement supports the need for improved communication methods, techniques and tools that can be used during the development process.
- In the current modern and pervasive computing environments, in which software development is outsourced offshore, the development approach should have a platform to enable different developers to share their understanding of the system requirements regardless of their location.
- The software product should be able to capture semantic and pragmatic (tacit) information in organizations.

- The social construction nature of information systems, as expounded by the three theories, AT, ANT and TOA, should be considered and given the highest priority.

- In addition to being able to model behaviour, the organizational culture and context should also be captured, as these are always an integral part of organizational behaviour.

- The functional requirements and the system requirements should be mapped from the organizational environment to the systems platform through a software model that does not neglect the social or human aspects of the organizational system.

- De Oliveira *et al.* (2006) state that a common repository, a guiding framework to the software process, domain and task knowledge are prerequisites for a sound environment for software development and should be captured in a software development environment (SDE). This repository is a store for all information related to the software development life cycle (SDLC). In addition, each software development process requires knowledge about the organization. This knowledge sets the context of the software product.

- Data and experience gained from previous software projects, in addition to identifying relevant key personnel that can work on a project should also be acquired, stored and reused.

- An analysis model, derived from the domain theory, should be designed for a family of systems in the same domain.

Lastly, this approach should completely ignore the reductionist tendencies of the structured, object-oriented and partly agile development approaches that currently guide our practice as software practitioners. The software development process needs to adopt the soft systems approach. More importantly, the methods used should be adapted to enable them to capture, store and maintain organizational behaviour in the software product. In conclusion, as this chapter has portrayed, a behavioural, socio-technical approach to software development should be the foundation for the development of romantic software products.

4.11 Summary

This chapter explored the softer issues of organizational information systems that are usually not considered in the hard sciences discipline such as computer science and engineering. To clarify the need for paradigm change, several theoretical frameworks were discussed, such as the theory of organized complexity, activity theory, actor-network theory and the theory of organized activity. These theories, together with the softer elements of organizations such as culture, context and people's behaviour, motivate for software development approaches that are grounded in the relativistic, interpretive or neo-humanist paradigms and that are able to capture, store and share with all stakeholders the knowledge found in organizations.

The discussion also highlighted the requirements of a software development approach. In this context an approach is at a higher abstraction level than a methodology. More so, an approach should be descriptive rather than prescribe methods to users, a task that is delegated to a methodology. A methodology, however, should comprise methods, techniques and tools that are used in a software development process. The content of this chapter will be used in Chapter 6 as additional and supplementary input to the data collected from software practitioners. It is, therefore, important for the reader to grasp that, although there is a myriad of attributes that can characterize organizations, solutions to capturing those described in this chapter have persistently eluded software practitioners. These will be at the focus of our proposed solution set. This discussion should be read in conjunction with Chapter 5, so as to enable the reader to understand the role that these two chapters play in this study. The conceptual grounding that supports the use of ontologies in software development will be discussed in Chapter 5.