CHAPTER TWO

System Development Re-Considered

This chapter starts from the concept of 'system' and the systems approach developed in 1960s which affected the thinking about computer system development. Then it turns to how a system is to be developed and the problems the developers, including software developers, always face when designing systems. The origins and key concepts of object-oriented methods will come next and how such ideas affect the process of software system development will be discussed. Finally the focus will be on the concept of circumscription, and in detailing the linkage between circumscribing and programming. Two paradigms, closed world and open development, which are related to this topic especially to the circumscription of knowledge, will also be discussed. This final section will also include the idea of 'evolutionary design' which is closely related to the EM approach. As this chapter covers work throughout the 1980s to date (the first three sections), and re-examines the issues in relation to circumscription (the last section), we call this 'system development re-considered'.

2.1 System Ideas and System Development

The idea of 'system' is commonly used in an informal sense as an abstraction from collections of real-world objects or human activities in the world[1], for example, the 'software system' or 'education system'. In this section we focus on the subject of 'systems': systems thinking, systems theory and system devel-

1. According to Harrington (1991), we talk of systems when referring to the completeness of a particular concept or to a process (e.g. "you cannot fight the system").

opment, then how systems theory and the use of systems ideas have been applied in organisations. Checkland's SSM is one example of an off-shoot of such work where systems thinking is a way of thinking about complex situations. Also we will try to find out how such systems theory and systems thinking have been applied significantly to computers and to the design process of software.

2.1.1 The Concept of 'System'

Today in technology we have been led to think not in terms of single machines but in terms of 'systems', especially as the more and more widespread use of computer systems has led to the increasing use of the term 'system'. Jordan (1968) lists fifteen different definitions of the term 'system' which cover a wide range of subjects, such as methodology, biology, physical and social sciences. But he concludes that

> a thing is called a system to identify the unique mode by means of which it is seen. We call a thing a system when we wish to express the fact that the thing is perceived/conceived as consisting of a set of elements, of parts, that are connected to each other by at least one discriminable, distinguishing principle. (p. 52)

This is the so-called 'core meaning' of 'system' and he comments that all these fifteen definitions are concrete exemplifications of this core meaning[2]. Jordan suggests that the term 'system' is used as an interaction between what is 'out there' and how we organise it 'in here', i.e. the interaction between the real world and how people observe or think about it[3].

Checkland (1993) says that a system has these properties: boundaries, inputs and outputs, components, structures, the ways of retaining its integrity, and "the coherency principle which makes it defensible to describe the system as a system". His discussion of systems leads to a new way of thinking about the world (i.e. systems thinking) based on which Soft Systems Methodology (SSM) was developed. The works of Boulding (1956) and Jordan (1968) were also the attempts at observing the world in systems

2. Checkland (1993) points out that the overall argument of Jordan's paper is that there is a core meaning of 'system' which makes it proper to attach it to many different things perceived in the real world outside ourselves.
3. To put it more simply, we use the term 'system' to describe to another person a set of elements and the nature of their relationships, so that the person can conceive of the set as a single, compound entity.

terms. The thinking of Checkland's SSM tries to find out how such systems ideas can help to tackle the unstructured problems which the conventional reductionist methods failed to address. Before further discussing the details of systems thinking in SSM, we should have a broad insight into systems theory and the approach of system development in general.

Systems Thinking

The concepts of 'system' have brought a big change in our views of the world, from the *mechanistic* view of the world by Descartes and Newton to a *holistic* view (Capra, 1996), that is, seeing the world as an integrated [functional] whole and not simply a collection of parts[4]. This holistic view in the twentieth century has led to a new way of thinking: *systems thinking*.

During the 1950s and 1960s, systems thinking had a strong influence on both engineering and management sciences, where systems concepts were applied to cope with *complexity* and solve practical problems[5]. Because the situations of business and science became more and more complex, scientists and managers had to concern themselves with not only the large numbers of individual components but also with the effects of the interactions of those components. Thus many scientists and managers began to analyse their strategies and methodologies using systems concepts. As Checkland (1993) describes:

> The systems engineer must also be capable of predicting the emergent properties of the system, those properties, that is, which are possessed by the system but not its parts. (p. 129)

Conventional Cartesian mechanistic thinking regards the world as a *machine* which can be understood completely by analysing its parts, i.e. the behaviour of the whole can be understood entirely from the

4. The emphasis on the parts is called mechanistic, reductionist or atomistic; the emphasis on the whole is called holistic, organismic or ecological (Capra, 1996).
5. In relation to complexity, Simon (1996) infers three 'eruptions' each refer to a different aspect of complexity: "The post-WWI interest in complexity, focusing on the claims that the whole transcends the sum of the parts, was strongly anti-reductionist in flavour. The post-WWII outburst was rather neutral on the issue of reductionism, focusing on the roles of feedback and homeostasis (self-stabilisation) in maintaining complex systems. The current interest in complexity focuses mainly on mechanisms that create and sustain complexity and on analytic tools for describing and analysing it". (pp. 169-170)

properties of its parts. Thus the method of *analytic thinking* (promoted by René Descartes) for coping with complexity is to break complex phenomena into different parts in order to understand the behaviour of the whole from the properties of its parts. In contrast, the main difference in a holistic perspective is the shift of emphasis from the parts to the whole. That is, the nature of the whole is different from, and even more than, the sum of its parts. Systems thinking embraces a holistic perspective but also emphasises *interaction* rather than just wholeness[6]. According to the systems view, the essential properties of a system are the properties of the whole and none of its parts have such properties, because such properties arise from the interactions and relationships among the parts. The properties of the parts are not *intrinsic* properties because the latter can only be understood within the context (or organisation) of the whole. In differentiating systems thinking from reductionist analysis, Capra (1996) characterises systems thinking as *contextual thinking* or *environmental thinking*:

> Analysis means taking something apart in order to understand it; systems thinking means putting it into the context of a larger whole. (p. 30)

The characteristic of systems comes from the properties of the whole (for example, the functionality or the purpose), which in turn arise from the configuration of relationships among the parts. The configuration of relationships is called a *pattern* which is also the focus of systems thinking in *cybernetics*[7]. Capra emphasises the importance of the idea of pattern as “the understanding of life begins with the understanding of pattern”. Systemic properties are properties of patterns, “what is destroyed when a living organism is dissected is its pattern”. The reductionist theories fail to cope with the issues of pattern because pattern is non-material and irreducible.

Checkland suggests that systems thinking is based on two pairs of core ideas: *emergence and hierarchy*, and *communication and control*. This shows another criterion of systems thinking: the ability to

6. Harrington (1991) characterises the holistic view as a perspective; whereas the systems view is a methodology.
7. The attention of pattern of organisation is the explicit focus of cybernetics (Simon, 1996; Capra, 1996), even though the development of cybernetics was independent of the one of general systems theory (both after World War II). Norbert Wiener, who inspired and invented the name of cybernetics, expanded the concept of pattern from the patterns of communication and control which are common to animals and machines to the general idea of pattern as a key characteristic of life (Capra, 1996).

shift our attention back and forth between systems levels (Capra, 1996). That is, in the systems view, different systems levels represent different levels of complexity and at each level the properties of the observed phenomena will not exist at lower levels. Such systemic properties of a particular level are called ‘emergent’ properties because they emerge at that particular level (Checkland, 1993; Capra, 1996). Further in the systems view, every ‘object’ is regarded as a network of relationships. Thus when we perceive objects as a network of relationships, what we observe are also an interconnected network of concepts in which *there are no foundations*. That is, the material world is a dynamic web of interrelated events and no properties of any part is fundamental. Such a multi-levelled structure is called a ‘hierarchy’[8].

The concept of *scientific objectivity* is also influenced by systems thinking, as Capra (1996) describes:

> In the Cartesian paradigm, scientific descriptions are believed to be objective, i.e. independent of the human observer and the process of knowing. The new paradigm [of systems thinking] implies that epistemology – understanding of the process of knowing – has to be included explicitly in the description of natural phenomena. (p. 39)

Because in a systems view the world is regarded as an interconnected web of relationships, the configurations of relationships (the patterns) are identified depending on different observers, methods and the processes of observation. Thus systems thinking means a shift from objective to ‘epistemic’ science (Capra, 1996); and the method of questioning is the main focus of systems theory. As Capra says: “what we observe is not nature itself, but nature exposed to our method of questioning”. In addition, that there are no foundations in the web of nature leads to the insight of approximate knowledge, i.e. this recognises that all the scientific concepts and theories are limited and approximate, and can never provide a complete understanding. This is in contrast to the Cartesian paradigm which asserts the certainty of scientific knowledge. Thus, according to Capra’s view, in science, we can only make limited and approximate descriptions of the described phenomenon.

8. Capra calls it ‘the web of life’ instead of a hierarchy to avoid misleading since the latter is derived from human hierarchies.

Systems Theories

Capra (1996) points out that systems thinking emerged simultaneously in several disciplines during the first half of the twentieth century, especially during the 1920s. The emergence of system theory, as Boulding (1956) describes it, came because of the increasing need for systematic theoretical constructs which can discuss the general relationships of the world, especially an increasing number of hybrid disciplines which connect with many different subjects[9]. Boulding presented the idea of *General Systems Theory*[10] (GST) which was used to "describe a level of theoretical model-building which lies between the highly generalised constructions of pure mathematics and the specific theories of the specialised disciplines". Conventional theories such as mathematics attempted to organise general relationships into a system, but a system may not have such connections to the real world. GST considers all 'thinkable relationships' abstracted from the concrete situation or knowledge of experience, and tries to make a connection between "the specific that has no meaning and the general that has no content". Based on the systems concepts that the world is a web of interrelated events and has no foundations for such a network of concepts, Boulding presents a preliminary hierarchy of real-world complexity. He suggests that the use of such a hierarchy can reveal the gaps in knowledge. He also argues we should never accept as final "a level of theoretical analysis which is below the level of the empirical world which we are investigating". This is why he puts in the title that GST is the skeleton of science because

> it aims to provide a framework or structure of systems on which to hang the flesh and blood of particular disciplines and particular subject matters in an orderly and coherent corpus of knowledge. (p. 208)

Earlier, von Bertalanffy (1968) introduced *General System Theory*[11]. By regarding the living organism as a whole, rather than simply a set of components with relationships between components, von Berta-

9. Boulding gives some examples such as cybernetics arising from electrical engineering, neurophysiology, physics, biology and economics; organisation theory arising from economics, sociology, engineering and physiology.
10. The name is slightly different to von Bertalanffy's (1968) *General System Theory* but Boulding attributes many of his ideas of GST to von Bertalanffy.
11. An earlier discussion of von Bertalanffy's ideas can be traced back to his paper: "General System Theory: A New Approach to Unity of Science", in *Human Biology*, Vol. 23, December 1951, pp. 303-361 (referred from Boulding, 1956).

lanffy shows an important distinction between systems which are *open*[12] to their environment and those which are *closed*. He emphasises that systems thinking is *process thinking* and based on this he formulated a theory of 'open systems' which combined the various concepts of systems thinking and organismic biology into the theory of living systems. As living systems have a wide range of shapes in phenomena, such as organisms, social systems or ecosystems, von Bertalanffy believes that a GST can offer the conceptual framework for unifying different disciplines[13]:

> General system theory should be ... an important means of controlling and instigating the transfer of principles from one field to another, and it will no longer be necessary to duplicate or triplicate the discovery of the same principle in different fields isolated from each other. At the same time, by formulating exact criteria, general system theory will guard against superficial analogies which are useless in science. (pp. 80-81)

Thus GST is regarded as a meta-disciplinary theory which abstracts from the special properties of specific disciplines and aims to provide a common language and theory for people in different disciplines to express and solve the problems of many different disciplines, especially with regard to the 'interdisciplinary movement'.

Systems Approach

There are two kinds of approach: the 'scientific approach' comes from science, just as 'systems approach' comes from the systems concepts. Checkland (1993) classifies both the scientific approach and the systems approach as meta-disciplines and both have a particular way to regard the world:

> The scientific outlook assumes that the world is characterised by natural phenomena which are ordered and regular ... and this has led to an effective way of finding out about the regularities – the so-called 'laws of Nature'. The systems outlook, accepting the

12.He calls such systems 'open' because they need to feed on a continual flux of matter and energy from their environment to stay alive.

13.According to Capra, before the 1940s the terms 'system' and 'systems thinking' had been used by several scientists, but it was von Bertalanffy's concepts of an open system and a general systems theory which established systems thinking as a major scientific movement. Checkland (1993) calls him the founder of the 'systems movement'.

> basic propositions of science, for it is a part of the scientific tradition, assumes that the world contains structured wholes ... which can maintain their identity under a certain range of conditions and which exhibit certain general principles of 'wholeness'. (p. 6)

The scientific approach involved reductionism, which tried to emphasise reducing the observed situation in order to increase the chance of experiment obtaining some observations which are reproducible. This approach assumes that the components of the whole are the same when examined singly, and the principles of assembling these components into the whole are also straightforward (von Bertalanffy, 1968; Checkland, 1993; Capra, 1996). These assumptions may be reasonable in physical science. But when moving to more complex phenomena such as social phenomena, it will be difficult to answer how to make the separation of the whole into 'components' or whether it is right to make such a separation. These can be also illustrated by the concepts of *restricted science* and *unrestricted science*. Physical science or chemistry is restricted science where a limited range of phenomena are needed to study, [reductionist] experiments in the laboratory are possible, and hypotheses which are expressed mathematically can be tested by quantitative measurements[14] (Checkland, 1993). By way of contrast, in an unrestricted science such as biology or social systems, the effects are so complex that experiments cannot be controlled easily or be precise. Quantitative models of physical science cannot be appropriately applied here and the effects of unknown factors (mainly from interaction and dependency of elements in systems) on our observations are also high.

The systems approach[15] aims to cope with the problems in unrestricted science by observing the world in a 'system way', i.e. as a set of systems. The following is a comment on the systems approach by one researcher in 1950s:

> Systems, of course, have been studied for centuries, but something new has been added. ... The tendency to study systems as an entity rather than as a conglomeration of parts is consistent with the tendency in contemporary science no longer to isolate phe-

14. Thus the greater the precision of the quantitative predictions and the more successful the experiments, the greater the confidence in the hypothesis.
15. Another reaction against the scientific approach is 'ethnomethodology' which reconciles a radical empiricism with the situatedness of social data, by looking at how members of a group actually organise their behaviour (Goguen, 1994 and 1996).

> nomena in narrowly confined contexts, but rather to open interactions for examination and to examine larger and larger slices of nature. (Ackoff, 1959)

Checkland (1993) argues that the process of the systems approach is similar to the one which occurs in human thinking, and the way we observe in systems approach is a world-view or *Weltanschauung*. “We attribute *meaning* to the observed activity by relating it to a larger image we supply from our minds”. The phenomena we observe (i.e. the essential properties) are only meaningful in the context of the whole: the *Weltanschauung*. As systems thinking has brought a new way of thinking in many scientific disciplines, many new concepts have been developed from the systems approach, even in restricted sciences in which conventionally reductionist experiment and quantitative measurements were thought to be enough to find out the principles. Examples include the quantum theory in physics. Conventional physicists believed that all physical phenomena can be reduced to some properties of elements (the material particles). However in the 1920s the quantum theory showed the fact that such particles can further dissolve at the subatomic level into wave-like patterns of probabilities (Capra, 1996). These patterns represent the probabilities of interconnections but not of the things. In such a view, our world is not regarded as a set of building-blocks, but as a complex web of relationships between the parts of a whole. Capra sums up the influence of systems concepts on quantum physics:

> Whereas in classical mechanics the properties and behaviour of the parts determine those of the whole, the situation is reversed in quantum mechanics: it is the whole that determines the behaviour of the parts. (p. 31)

Systems Classes

There are many similarities between the two systems ontologies[16] of Checkland (1993) and of von Bertalanffy (1968). The following summarises Checkland’s systems classes, which are mainly influenced by Boulding’s hierarchy and Jordan’s taxonomy[17] in their attempts to survey the whole of the real world in systems terms.

16.According to von Bertalanffy, ‘systems ontology’ refers to the kinds of system that there can be.

17.By analysing these works, Checkland justifies that the four is the absolute minimum number of systems classes need to describe the whole of reality.

Natural Systems Checkland defines natural systems as the systems “whose origin is in the origin of the universe and which are as they are as a result of the forces and processes which characterise this universe”. He further says that natural systems are evolution-made and irreducible wholes[18].

Designed Physical Systems These are, the systems which are man-made and are the result of human conscious design[19]. Both natural systems and designed physical systems can be categorised as *real systems* in von Bertalanffy’s term which means “entities perceived in or inferred from observation, and existing independently of an observer”. The main difference between natural and designed physical systems is their origins. Designed physical systems exist because they are needed for the human activity system (which will be described later).

Designed Abstract Systems This is similar to *conceptual systems* in von Bertalanffy, which means the ordered conscious products (symbolic constructs) of the human mind, such as logic, mathematics, poems or philosophies. Checkland argues that “they will exist as a result of a positive act related to some objective – elucidation, the enlargement of knowledge, or an inner urge to express the inexpressible”.

Human Activity Systems The concept of human activity systems is that of a number of human activities related to each other which can be viewed as a whole. Checkland argues that the fact these activities form a human activity system is emphasised by the existence of other systems (often designed systems) which are associated with them[20]. The main difference between natural systems and human activity systems is that in the latter we cannot establish *full* public knowledge because the entities perceived are not independent of an observer, as in natural systems[21]. Thus reductionist methods which are suitable in restricted science may not be appropriate for unrestricted science.

18. For example, according to Checkland (1993), carbon dioxide is not reducible to carbon and oxygen, we can even find its interatomic distance and bond angles.
19. Sometimes we call them *artificial systems* because they are designed to achieve some human purposes.
20. These “other systems” take on a quite different character when they are viewed as part of a human activity system, because of the impact of the human intervention.
21. That is, according to Checkland, that human free will (i.e human freedom of choice in selecting actions) means that the principles and logic of natural systems cannot be totally applied to human activity systems.

'Social Systems' People in a group (i.e. a social system) have their responsibilities as well as having some expectations of other members. Checkland suggests social systems can be placed between natural systems and human activity systems. Because social systems observed in the real world should be a mixture of a rational assembly of activities (the human activity system) and a set of relationships (the natural system). The important aspect of social systems is that human beings are components which are active participants in social phenomena. As with human activity systems, we cannot observe and explain human activities in the same way as natural systems. Also we cannot predict what will happen as such systems consist of both intended and unintended effects or they may even be influenced by the growth of human knowledge that is unpredictable as well[22]. By contrast with finding 'laws of Nature' in natural phenomena by scientific approaches, in social systems we can only reveal the *trend* rather than *laws,* and reduce the observed subject only to what Checkland refers to as the *logic of situations* rather than social reality[23].

Checkland's Soft Systems Methodology

Today systems thinking has been the paradigm of thinking holistically and has much influenced the later development of systems ideas as well as the development of the use of systems ideas in particular disciplines. SSM, as Checkland describes, is the combination of these two developments and aims to see if systems ideas can help to tackle the problems of 'management'[24] (the broader term for social phenomena). He summarises four reasons for using the systems concept in management problems:

> Firstly, many problems are seen to recur when they are looked at as problems of systems; hence solutions may be transferred. ... Secondly, the systems view enables problem-solving to concentrate on the *processes* by which things are done, rather than on (*ad hoc*) 'final outcomes'. Thirdly, systems may provide the objective standard by which problems can be organised for solution. ... From objective standards we may be able to

22. Checkland also points out that not only complexity in social systems but also the non-availability of experimental objects will be problems.
23. Over a long time this logic of situations will increasingly change because of the growth of human knowledge and humans being involved.
24. That is, the real-world problems or the problems of decision in social systems.

> gain greater insight to generalise on business phenomena. Fourthly, ... many business problems are 'mixed', that is they have both qualitative and quantitative attributes. Systems analysis is the newest technique to be brought to these mixed problems. (Checkland, 1993, p. 147)

The early experiences of SSM in trying to apply the systems approach outside technical areas can be found in his book *Systems Thinking, Systems Practice*[25] (*STSP*, Checkland, 1993). In connecting with the four systems classes mentioned above, Checkland (1993) summarises the following would-be problem solving approach:

> Firstly, there will be much to *learn* from natural systems. ... Secondly, the problem solver is free to *use* designed systems, whether physical or abstract, to achieve his ends. And thirdly, ... since human beings are purposeful, that we may seek to 'engineer' human activity systems, using the word 'engineer' in its broadest sense. (p. 125)

The starting point of SSM is to consider an organisation as a system with functional sub-systems (such as production, finance, human resources, etc). The real-world experiences can enable us to build the knowledge of these [sub-]systems and support a better design and operation of systems in real situations. Similar to systems thinking, the process of SSM is a *learning cycle* by which the tentative ideas can be applied to inform the practice which then becomes the source of richer ideas. Then these ideas and the ways of using such ideas can be extended as a result of practical experiences. The development of SSM as a learning process is based on the following key thoughts (Checkland, 1999):

- Every situation in the research is a human situation in which people are attempting to take *purposeful* action which is meaningful for them. This thought leads to the idea of modelling purposeful human activity systems: the sets of human activity "which together exhibit the emergent property of purposefulness".
- Because of the complexity of human activity systems, there may exist many models for systems interpreted for different purposes[26]. Checkland suggests that the first choice of models should be

25.'Systems practice' here implies a way to find out how to use the systems concept to solve problems.

the one that is the most relevant in exploring the situation. Then we decide the perspective or viewpoint for each activity, i.e. the *Weltanschauung*, based on which model will be built.

- Similar to the various interpretations of purpose, we may find many models to represent the situation. This makes SSM different to conventional systems engineering. SSM thus is regarded as an inquiring process as it is not working with the 'obvious' problem to find out solutions. In contrast, it works with the idea of a *situation* which people may regard as problematical for various reasons. SSM in this case is regarded as an organised *learning system*.

Checkland emphasises that the term 'system' in SSM is not applied to the world but instead applied to the *process* of dealing with the world[27]. This is different to a conventional approach of systems engineering which regards the world as a set of interacting systems, some of which can be engineered to make them work better. This also frames the distinction between two kinds of systems thinking: the latter is 'hard systems thinking' and SSM is 'soft systems thinking'. Their difference is illustrated in Figure 2.1. Harrington (1991) says that the hard approach views a system as a machine with definite and identifiable input/output processes, and the system here is defined as a way of getting things done. Therefore this approach emphasises the 'doing' aspect of systems methodology and thus on the physical domain. In contrast, the soft approach is organically-oriented because many aspects or elements, which make up the system, are not easily controlled and defined, and the interactions among them depend on circumstances. It is not only linked with the physical domain but also the perceptual domain. Wholeness and relationships are important because the soft system only has meaning as a whole, i.e. the parts have meaning only when relating to the whole. As the interaction of parts is essential for the systems as a whole, the emphasis of the soft approach is on the maintenance of stability. To summarise, the hard approach is prescriptive analysis whereas the soft approach is descriptive analysis (Harrington, 1991).

26.When modelling purposeful activity, many interpretations of 'purpose' can be possible. Checkland (1999) takes the Concorde project as the example. It is commonly regarded as an *engineering* project to produce supersonic aircraft (technical world-view). But it is also treated as a *political* project because at its origins President de Gaulle of France was vetoing the British entry into the European Common Market and this project is the result of negotiation (political world-view).

27.This idea is similar to von Bertalanffy's as he emphasised systems thinking as process thinking.

In the learning process associated with SSM, the models used are *devices* for helping to structure an exploration of the problem situation. This is different from the models used in hard systems thinking as they represent the parts of the world outside ourselves. Checkland (1999) makes these comments on the models in SSM:

> They are not models *of* ... anything; they are models *relevant to debate* about the situation perceived as problematical. They are simply devices to stimulate, feed and structure that debate. (p. A21)

To conclude, SSM presents a more complex interpretation which does not readily give us solutions. It attempts to describe, so as it enables us better to understand the situation rather than prescribe.

Computer 'Systems' and Software 'Systems'

Today computers are argued to be more complex than most things people build because of their large numbers of *states* (Brooks, 1987). This makes the conceiving and developing of computers become dif-

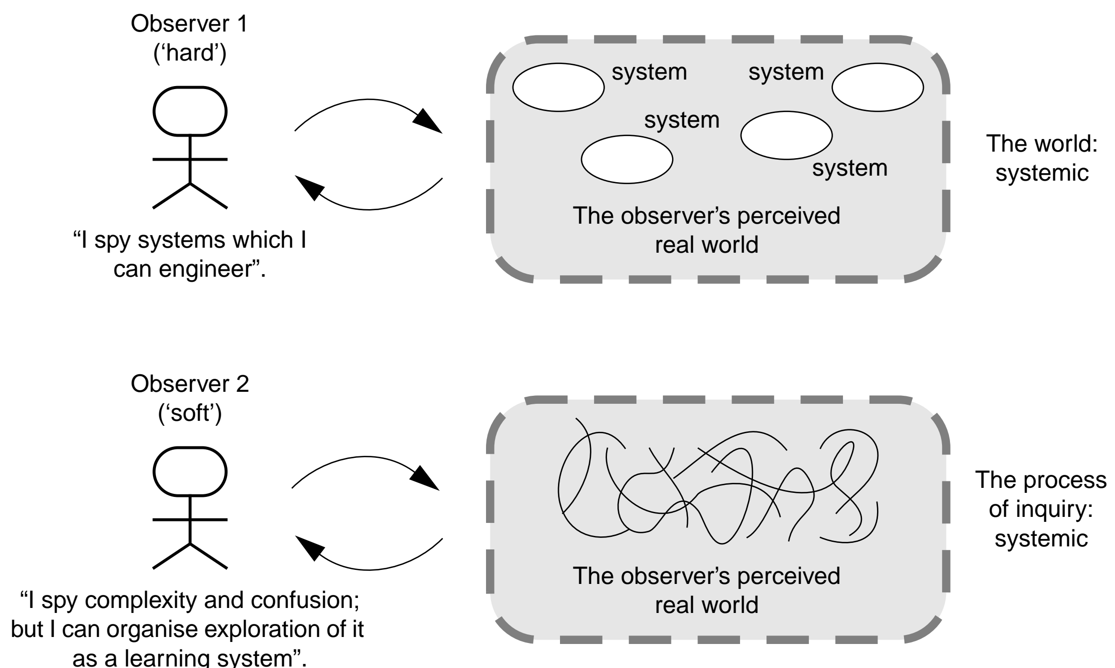

Figure 2.1 The Hard and Soft Systems Stances (adapted from Checkland, 1999)

ficult. However, the model 1401, announced by IBM in October 1959, was not so much a computer as a computer *system*. At that time many computer companies were obsessed with the designing of CPUs[28] and thus did not consider the system to be designed and its external context as a whole[29] (Campbell-Kelly and Aspray, 1996). However IBM recognised that customers were more interested in the solution to business problems rather than a solution for computer design. Thus they started on the total-system view of computers, i.e. the systems thinking, which considered programming, customer transition, service, logistics, etc. This change of thinking, as Campbell-Kelly and Aspray comment, brought the success of IBM 650 and was the factor that most differentiated IBM from its competitors.

It is commonly recognised by most software programmers that the greatest difficulty of writing software is that the programmers have to anticipate all possible outcomes of computers. Brooks (1987) considers the difficult part of writing software is the development of the conceptual construct. He points out four essential[30] difficulties in software technology:

- **Complexity** The complexity of software comes from the increasing numbers of different elements, not only merely repeating the same elements to large size. As the interaction of such elements is non-linear, thus the complexity of the whole software increases more than linearly.
- **Conformity** For conventional [restricted] sciences, scientists have a faith that there exist unifying principles and which can be found[31]. However no such faith for software programmers because, according to Brooks, software is designed by human beings (and not by God), and the interfaces or structures of the software may differ at different times.
- **Changeability** Software is more constantly changeable than manufactured products in general engineering. Not only because software can be changed easily as it is a pure conceptual con-

28.Central processing units.

29.It was common at that time to regard processors as 'all the rage'.

30.Brooks names *essential* here, following Aristotle, the difficulties inherent in the nature of the software which are also essential properties of software, while as *accidental* means those difficulties which attend its production but are not inherent.

31.Brooks reports that "Einstein repeatedly argued that there must be simplified explanations of nature, because God is not capricious or arbitrary".

struct, but also the function or behaviour of a system is decided by software, and generally system function is the most changeable part because of some pressures[32].

- **Invisibility** Software is invisible. Thus it is hard to find a proper representation for it. Diagrams are commonly used to represent software structure, but sometimes we need several diagrams to represent different aspects of software such as dependency or time sequence. Brooks makes the criticism that structures of software still remain inherently unvisualisable which “not only impedes the process of design within one mind, it severely hinders communication among minds”.

The use of the term ‘software systems’ may be traced back to the 1960s, when the software programmers found the difficulty of exploiting the potential of the rapidly developing hardware and the difficulty of managing big projects which were going disastrously wrong. The conference entitled ‘Software Engineering’ held in Garmisch, Germany in October 1968[33] marked the first time software development was regarded as a engineering discipline[34]. From this point on, systems thinking had a prominent influence on software development. Due to the difficulties of software construction mentioned above, people found that systems concepts could be applied with conceptual integrity to cope with the complexity in problems, especially those whose solution involves the combination of software and hardware. As the requirements of software escalated rapidly, programs became ‘software systems’ (Beynon and Russ, 1994). Today, software plays such a critical role in organisations that the term ‘software systems’ is commonly used in both engineering and management, and has even become a discipline. The introduction of systems concepts in software due to the development of the software and the design of organisational structures are mutually dependent. They cannot be developed in isolation and also they affect each other. Warboys et al. (1999) conclude that “a software system in its real world domain of application can be thought of as a system within a system”.

32.Such as users wanting to extend its functions or due to new devices of computers.

33.Campbell-Kelly and Aspray (1996) say that “the real importance of the meeting was to act as a kind of encounter group for senior figures in the world of software to trade war stories”.

34.Brooks (1995) comments that the goal of applying engineering principles to software production in the 1970s was to increase the quality, testability, stability and predictability of software products, not necessarily the efficiency of software production.

2.1.2 Software System Development

Software system development is a process in which system developers design, implement and maintain the software and models of computer systems[35]. Traditionally computers were designed to replace some other devices or skilled operators for the sake of efficiency and effectiveness. In business, computers were developed to analyse and automate the existing data-flows and databases which means running the business processes that were paper-based before. Harel (1992) calls this 'one-person programming' because such requirements are known in advance and are straightforward relative to the construction of software. Structured methods for system development were first developed in the 1970s which gave systems developers useful ways to build effective requirements during the late 1970s and 1980s. However as the requirements escalated, the problems of the *software crisis* emerged because systems took too long and cost too much to develop, and even after that they were not working as we expected. Furthermore, there exists a problem between the rapidly growing software code and the shortfall of personnel to maintain the code. According to O'Callaghan (1994), maintenance costs for legacy systems are reaching 90% for the total cost of software. Thus other methodologies, such as object-orientation, to software development were advocated to deal with the software crisis.

The Systems Boundary

It is important to define the system boundary while developing a system. The boundary is defined by Checkland (1993, p. 312) as "the area within which the *decision-making process* of the system has power to makes things happen, or prevent them from happening", or more generally, "a distinction made by an observer which marks the difference between an entity he takes to be a *system* and its *environment*". Generally inside the boundary are the system itself and other elements the designers intend to control; outside the boundary is the environment with which the system will interact. Regnell (1999) designates the former the *target systems* and the latter the *host system*. However it is not easy to define

35.We use the term "system development" here to mean "software system development" which focuses on the functionality or behaviour of computer systems, i.e. on *what* the system does rather than *how efficiently* it performs.

the system boundary as we need to decide which elements will be either within the system or a part of the environment. This is a issue of *circumscription* which will be focused on later in this chapter.

Cook and Daniels (1994) identify three categories of software system boundary: (1) *hard* system: the system boundary is explicit and the functionality or behaviour of the system is totally dependent on its software; (2) *semi-hard* system: the introduction of software will not change the behaviour of the whole system, but the role of software depends on non-technical factors such as cost or political issues; (3) *soft* system: the introduction of software will consequently impact the system and the environment in unpredictable ways, thus later we need to change the software in order to reflect and incorporate such change. We may regard the systems developed in a conventional way as 'hard' systems because their boundary is defined in advance and the system behaviour is perceived as totally dependent on software. Systems developed in the EM way may be regarded as 'soft' systems because their boundary is not explicitly fixed and is changing all the time. Thus the behaviour of systems and their interaction cannot be predicted and this is why we are focusing on observation and experiments in the EM models. 'Semi-hard' systems are more related to information systems as we would like to automate some operations and processes in an organisation but hope this will not impact the overall organisational operation. Thus traditionally the introduction of information systems into organisations is on a piecemeal basis and we assume that the software does not affect the basic structure of the organisations. However in many cases the introduction of information systems are changing the overall organisational processes, and this is the main issue BPR is focusing on. This issue will be further discussed in section 3.3.

System Development Process

Morris et al. (1996) suggest that system development during the 1970s was moving towards *engineering* techniques rather than *programming* or *management* techniques. That is, it was based on graphical models which represented various views of the proposed system ('engineering techniques'), rather than carried out by the programmer without any external reference ('programming techniques') or supported

only with textual specifications and documentation ('management techniques'). Warboys et al. (1999) consider the term 'engineering' from a more general viewpoint not confined to technical sciences:

> *Engineering* is the application of scientific principles to the solution of real-world problems. It is not about finding the truth, it is about intervening in the real world with artificial constructs intended to ameliorate some condition otherwise affecting people. It is about making best of the knowledge that we have, albeit imperfect, incomplete and uncertain, and finding safe, pragmatic and economic solutions within known constraints. ... The engineering of processes is a way of solving problems arising in the context of an organisation, in particular those organisational problems whose solution lies in exploiting the use of coordinative and integrative technologies. (p. 55)

In relating to systems thinking, the way we observe the world is a process of questioning and through which the limited and approximate knowledge is obtained[36]. In summary, engineering as systems approach relates to problem-solving via the construction of models (i.e. the artificial artefacts). The process of engineering is mainly based on personal experiential knowledge acquired from much trial and error, learning from observing which solutions work or fail. Each solution will be the model for the next stage. The process of engineering may also change people's behaviour. Sometimes we say *system engineering* or *software engineering* which relates to software development mainly focusing on solving problems in software design. In general the process of software development consists of three essential stages:

- **Requirements**: requirements elicitation, capture and approval (*what* functionality/behaviour the system should have);
- **Design**: study the requirements and design a solution (*how* the specific functionality, defined in requirements, to be built);
- **Implementation**: implementation, coding, testing and maintenance.

Figure 2.2 shows an integrated view of software development process which starts from requirements elicitation and analysis, through design, implementation and testing phases, finally to validation and cer-

36.Because there are no foundations in the web of our life.

tification before releasing the system. Regnell (1999) says that the development process starts with the external view of the system, where the system functionalities are defined in requirements specification. It then continues to the design and implementation stages which focus more on internal system issues, and finally it returns to the external view for validation. The external view (or *black-box* view) focuses on external functionality as observed by the users of the system. The internal view (*white-box* view) focuses on system architecture as well as the structural (and behavioural) properties of elements in the system[37]. There are also two views for software testing: black-box testing means testing system functionalities whereas white-box testing refers to structural testing. When the system has been implemented, its function is determined by its program.

With reference to Checkland's characterisation of scientific and systems approaches, there are a number of reasons why the conventional system development process can be categorised as a scientific approach. It is based on the subdivision of the development process (i.e. requirements, design and

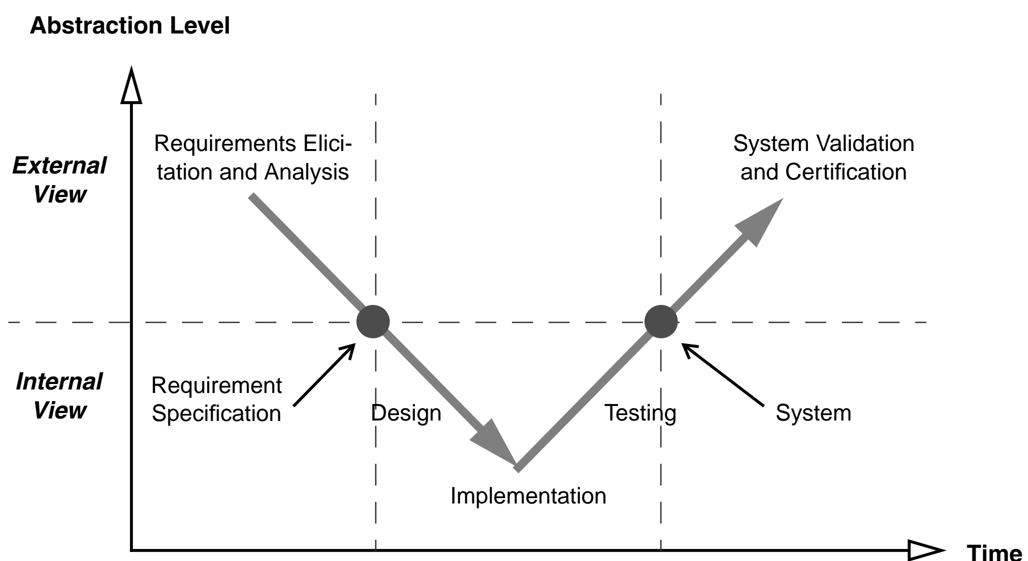

Figure 2.2 Different Abstraction Levels over Time (adapted from Regnell, 1999)

37.Checkland (1993) has a similar viewpoint in SSM – he classifies the perspectives of the observer on the behaviour of systems in two ways: either he can concentrate exclusively on the inputs and outputs and in this case treat the systems as 'black-box'; or he can describe the internal state of the systems in terms of variables, or the path of state changing influenced by external conditions.

implementation). It can also be viewed as a linear-phased process, even if sometimes there is an iteration across these phases for enhancing specifications until systems are released. On this basis, both the sequential 'waterfall' approach and the iterative 'spiral' approach will be categorised as scientific approaches. The development process is fragmented into different procedures and the completion of each phase is prerequisite to the beginning of the next one. The system to be developed is viewed as a machine with definite and identifiable input and output processes. The emphasis of such approaches is on the 'doing' aspect (i.e. getting things done) and the domain is mainly the physical domain[38]. One of the reasons, as described by Loomes and Nehaniv (2001), that software engineering has been dominated by such a lifecycle model of design is that "the design problems are still formulated with respect to a single, reified, pre-supposed system, referred to, already before the fact, as 'the system'". In their discussion of the reification of 'the system'[39], they point out that:

> Within this [life-cycle] model, there is the assumption that 'the system' exists in some embryonic from the very start of the process, and that the task of the designer is to nurture development and allow maturity to be reached in good health. (p. 27)

The common feature of these approaches is assuming the development is a one-time effort (Lehman, 1991) and the modification and enhancement to the systems are treated as a new or different development process. This may result in high maintenance costs and inflexible systems which are difficult to adapt to changes in the dynamic business environment. And the sequential subdividing process in the development may restrict the designer's response to dynamic situations. Further the systems development, as well as modification and enhancement, may take a long time and be difficult to carry out because of the 'frozen' functionality of the systems. The same task may be classifiable in two different phases, e.g. the software changes in testing phase may not be different from the changes in the maintenance phase. Such problems have led many companies to adopt a non-structured approach, such as

38.This is in contrast to the systems approach in which the aspects (or elements) or a system are not easily controlled and defined (because the interactions among them depend on circumstances). The emphasis is on the maintenance of stability (cf. SSM) and the domain includes both the physical and perceptional domain.

39.They say that concepts of 'the system' are treated as an entity with a single identity whose existence is tacitly accepted and validated in unspoken implicit assumptions. For example, 'the system' as discussed by designers, the representation or prototypes of 'the system' in its development, 'the system' as a physical, deployed entity, or the maintained or modified versions of 'the system'.

object-orientation, in their system development. But even in object-orientation, the development and maintenance are still treated as two different activities.

Structured Analysis and Design

Structured methods for system development were widely used during the 1970s and 1980s. They took the reductionist view that the best way to manage complexity was to limit the designer's view of the domain. That is, a program should be modular (i.e. divided into many small parts) so that designers could write a module with little knowledge of the code of other modules. The designers begin with an overview of software artefact and, when this has been satisfactorily developed, their attentions shift to a lower level of detail. The behaviour model (or functional model) for the proposed systems is built via a decomposition of functionalities which leads to a network of concurrent processes. The claim is that modularisation is a mechanism for improving the flexibility and comprehensibility of systems and shortening the time of development[40]. Structured methods focus on the processing aspect because traditionally people believe that the systems are designed to perform work. Structured methods, reflecting a reductionist view, make a distinction between data and processing, which enables the designers to observe the problem in terms of three dimensions: the processing, the information structure and the time sequence[41], and then integrate these views into whole systems. Some programming languages, such as COBOL, C or Pascal, promote/encourage such distinction.

There are two main paradigms for system development: *waterfall model* (Figure 2.3) and *spiral model* (Figure 2.4). The waterfall model decomposes the developing process into sequential phases, and the sub-process of each phase is totally dependent on the results of the previous phase. O'Callaghan (1994) says that because the structured analysis decomposes the functionality of proposed systems into small ones, waterfall model is thus the main paradigm used in these conventional methods.

40. Parnas (1996) asserts that the effectiveness of such modularisation depends on the criteria of dividing the system into modules.

41. Sully (1993) gives some examples of tools focused on each dimension: the data flow diagram (DFD) addressed the processing; the entity relationship diagram (ERD) addressed the information structure; and the state transition diagram (STD) addressed the sequence behaviour.

He further argues that the main problem in the waterfall model is its application-centred character. That is, the overall functionality of the application has to be defined firstly in order to define other small modules. Also it may become difficult to define and analyse the functionality for complex systems, and if errors made in the early stage, it will be very expensive if such errors are not found until all the software has been developed. Spiral model may sort out such problems but still have some disadvantages when applied to structured methods.

System development using structured methodologies raises some problems. Structured analysis defines the requirements from three dimensions (processing, information and time) from which it may be easy for designers to construct specifications. However systems developed in this way may be resistant to change and extension. The reason is such methods focus on the *processing* rather than the *concepts* (or objects) of the system (Sully, 1993). Sully criticises that processing of a system is the most volatile aspect of a system which may change rapidly over time. In contrast object-orientation focuses on 'concept' and aims to build systems more resilient under change[42]. Other potential problems may occur from transformation between requirements specification to implementation. As Morris et al.

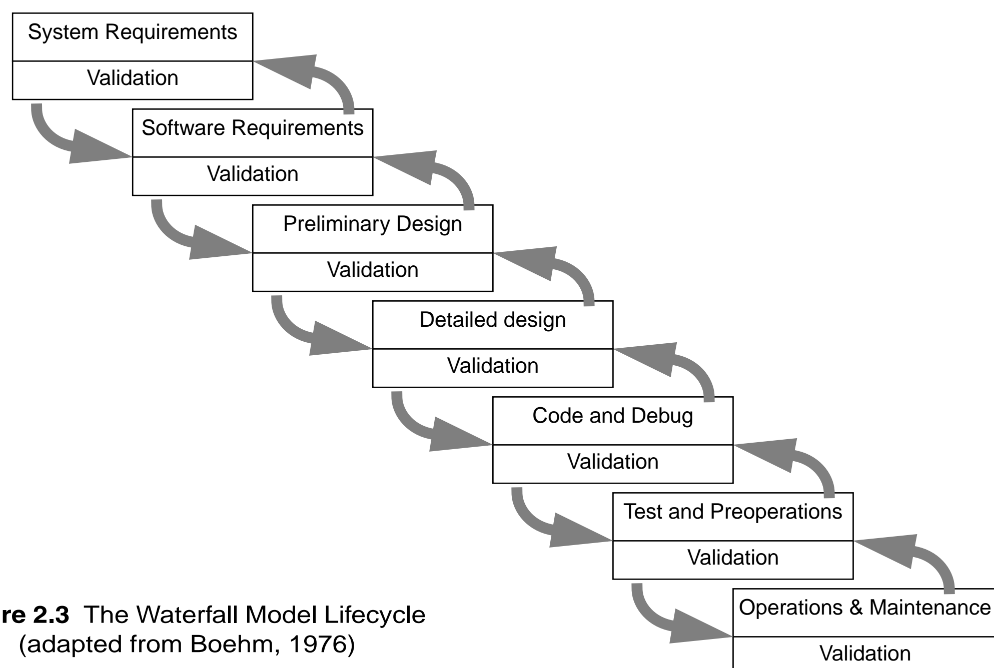

Figure 2.3 The Waterfall Model Lifecycle (adapted from Boehm, 1976)

(1996) describe, the weakness of all structured methods is that the effort of analysis may be left behind, just when it has reached the peak of its development, and the development is going into the implementation stage. That is, a new structure will be defined in implementation which tries to map the products of earlier analysis to some executable functions. Error is likely to occur during this transformation and it will become worse if such error occurs at later stages[43].

2.2 Object-Orientation

The object-oriented languages have been around since mid-1960s and became popular in academic research in the 1980s. But the realisation of object-orientation in both industry and business has started from the 1990s due to the reduction of memory cost and increasing processing power. Another reason is that systems developed by OO methods have shown themselves to be more resilient under change. Today object-orientation applies not only to tools and a way of thinking in computer programming, but its

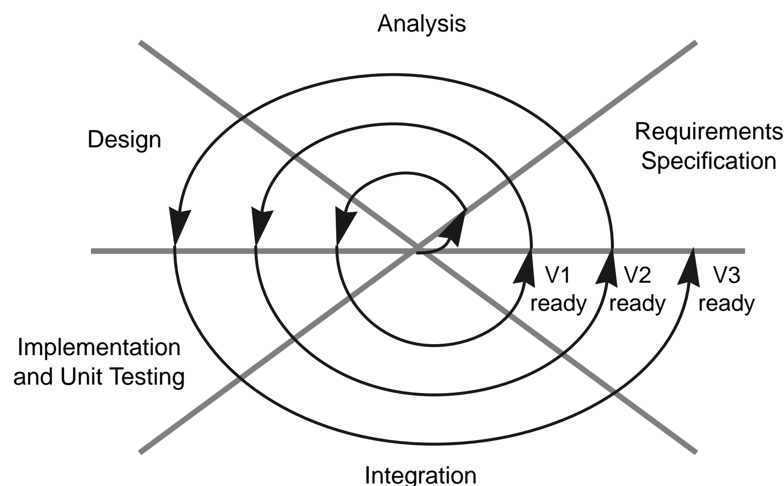

Figure 2.4 The Spiral Model Lifecycle (adapted from Boehm, 1981)

42. In OO, all the processing units are organised around the object concept, and the detailed design of processes can be done by implementing objects.

43. OO methods try to avoid such problems by ensuring that the objects identified in analysis phase will evolve to the objects of implementation.

concepts are also commonly used in the information processing industry (i.e. information technology) and BPR. This section summarises the general issues of object-orientation and describes the influence of object-orientation in system development.

2.2.1 The Origins and Key Concepts of Object-Orientation

It is widely accepted that the original ideas of *objects* came from Simula which is regarded as the first object-oriented language (Sully, 1993; Graham, 1994; Cook and Daniels, 1994). Simula first introduced the concept of *classes* and *inheritance* to programming languages and the main purpose for which Simula was designed was expressing discrete-event simulations. The influence of Simula in object-oriented languages can be seen today in the way that OO models aspire to mimic the real world and to map the behaviour of parts of the world. As Sully (1993) remarks, OO solutions are viewed as simulations of the real world, and this principle was 'discovered' when using Simula. These core motivations for object-oriented modelling are also represented in the EM approach to be introduced in chapter 4. When we perceive solutions via computer models which mimic the real world, the language we use can reflect our mental model of proposed systems. When we simulate real world subjects, we can *observe* the effects of different parameters rather than *calculate* them[44].

Object-orientation became popular both in industry and business and widely discussed in academia in the 1990s. The reasons can be summarised as following:

- **Software Crisis** The software crisis is the problem discussed in subsection 2.1.2 which emerged because of the rapidly growing volume of software and the shortfall of development personnel to maintain the code. Further it becomes more and more difficult to meet the user's requirements in software quality as conventionally we only highlight the expectations of system functionality. OO is claimed to be a good solution for the crisis because it provides an environment

44.For example in simulation we can observe the car behaviour when passing an intersection via queue lengths and time delay.

for higher productivity and makes software more reusable, which requires less effort for development and maintenance.

- **'Natural' Abstraction** The term *natural abstraction* (O'Callaghan, 1994) means a higher level of abstraction which can be regarded as a common 'language' between users and designers. Thus the natural abstraction can map easily from the user's domain to the designer's domain. The concept of objects can be the high level of abstraction through which designers can structure their concepts easily and use such abstraction to model the problems. At the same time, designers and users can 'talk' to each other in this common 'language'. O'Callaghan concludes that "as the structuring concepts are chosen from the *problem domain* rather than imposed on the problem from the domain of computer science", this makes OO more natural than structured methods.
- **Reuse of System Components** It is desirable that systems should be developed as economically as possible with the minimum duplication of effort due to the software crisis. For this reason a library of reusable components is required. The objects in OO which bind the processing and data in a unit can be the reusable components of system development. Also the *design patterns* for software architecture are another example of reuse in system development. They constitute a 'grass roots' effort to build on the collective experience of skilled people (Buschmann et al., 1996). As such experts have already developed solutions to recurring design problems, using these design patterns can help to base the development on proven solutions of standard subproblems.

Sully (1993) identifies three main characteristics of object-oriented language: *object-based* means the OO language has the facility for encapsulation and to represent object identity[45]; *class-based* means the facility for offering an object-based unit and abstraction; *object-oriented* means the facility to offer class-based units inheritance, self-recursion and a higher degree of object character. The term *encapsulation* – that data and the operations performed on it are combined together (encapsulated) into objects – means that OO can provide a boundary around the object which can hide the complexity (i.e data and methods) by limited visibility and information hiding[46]. This can help designers to concentrate

45.Gammack (1995) comments that two concepts, encapsulated objects and messaging, suffice for a system to be called object-based.

on the conceptual view of objects without considering how to implement the code or other physical realisation. The *inheritance* mechanism in OO languages allows the specification of subclasses to inherit data from the higher-level class. In this way the designers have more flexibility where they need only to recognise the general and the specific parts then add the extra behaviour.

The Unified Modelling Language (UML) is the synthesis of object design languages. UML merges Booch notation, Object Modelling Technique (OMT) and Object-Oriented Software Engineering (OOSE) and was released in 1997 by Object Management Group (OMG) for a standard object-oriented methodology[47]. UML includes a set of consistent diagrams which can be used to describe the systems requirements, design and implementation. It seems to be more complete than other OO methods in support for the modelling of complex systems.

The Difference between Structured and OO Methods

Object-orientation can be viewed as initiating a totally new paradigm for programming compared with structured methods. In general, OO methods are *structural* and *problem-oriented* approaches whereas structured methods are *functional* and *solution-oriented* approaches (Graham, 1994; Morris et al., 1996). The following summarises some issues raised while discussing the difference between OO and structured methods.

- **The Paradigm Shift** The first difference is in the perception of designers when they start to develop the software. Through OO methods their thinking is on how to design the objects which can be reused many times, in comparison to the structured methods where their thinking is on how to design the specifics for each function of the proposed system. Thus OO methods are problem-oriented because designers need to understand the problems well in order to design suitable objects. It is also reuse-oriented because at first the designer thinks of the generalised units and

46.Or we can say each object in the system is acting as a 'black box' which only showing limited visibility (but not internal details) to external observers. The object defines a visible interface which declares to other objects (i.e. its client objects that use it) what operations on the hidden data are recognised.

47.It was submitted by Rational's Grady Booch, Jim Rumbaugh and Ivar Jacobson, as well as other major companies which include i-Logix, Digital, HP, ICON Computing, Microsoft, MCI Systemhouse, Oracle, Texas Instruments and Unysis.

based on these many subsequent applications can be developed. Table 2.1 shows three differences between these two paradigms.

Old Paradigm	Object-Oriented Paradigm
Separate data from processing.	Integrate data and processing.
Build distinct applications from specifications.	Place each part at its correct place within the whole.
Design processing by top-down functional decomposition.	Design processing within the context of an object model.

Table 2.1 A Comparison of Paradigms (from Cook, 1994)

- **Modules versus Objects** The concepts of modules and objects may be similar but their characters are quite different. In structured methods, the main function of proposed systems is decomposed into many sub-functions which are allocated into modules. Thus modules are interdependent on each other and most of them are not usable outside the system. The main function of the system is built up by these modules in a hierarchical structure. It may take a short time to develop such modules initially but maintenance may become difficult to control and too expensive in large systems. In contrast, the objects in OO methods can localise change within one unit by information hiding[48] and so promote reuse. The design of suitable objects may take more time initially because it involves thinking about the use of objects in many contexts, but later it will become easier and more effective to use objects. In this way the object-orientation can encourage us to take a wider view of system development. Figure 2.5 shows the different efficiencies among unstructured, structured and OO developments along with the lifecycle of software development and maintenance.
- **Scientific Approach versus Systems Approach** We have already discussed the scientific approach and systems approach in subsection 2.1.1 (under 'Systems Approach"), and focused on their different ways of problem solving. OO is problem-oriented because designers have to

48.O'Callaghan (1994) views the OO systems as a network of co-operating 'virtual machines' because the behaviour of each machine is defined by its protocol of operations, rather than just a single, subdivided machine with the main function at the top.

understand the problems (i.e. the whole context) in order to design reusable objects. Thus the OO development is facilitated by a whole approach and Sully (1993) also has a similar viewpoint. He points out that the key difference between OO and structured approaches is that the designer should consider all the enterprise concepts which are likely to be involved in order to define objects for the application. In conclusion we may regard the OO approach as a kind of systems approach because we take an enterprise-wide view of system development. This is in contrast to structured methods as scientific approach because its emphasis is on the decomposition of system functionality and the construction of modules – a set of independent system components, and its development is generally a linear-phased process. As each module is to perform a single function, any change to that function requires a change in only one, identifiable, part of the system (Crowe et al., 1996).

2.2.2 *The Claims and Problems of Object-Orientation*

Object-orientation is claimed to be the most natural way to understand and model the problem domain in which a system will operate. The whole idea of OO is that the object is an abstraction which more closely mimics the real world than conventional structures. Further, the encapsulation of process and

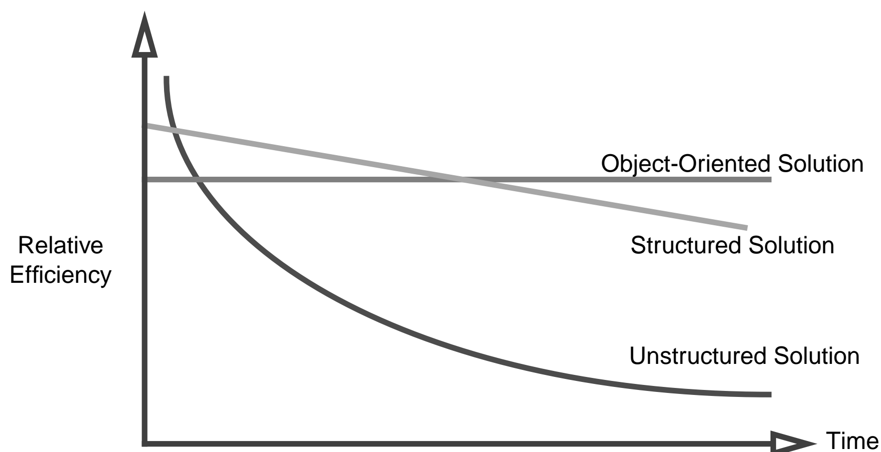

Figure 2.5 The Relative Efficiencies of Unstructured, Structured and Object-Oriented Development (adapted from Sully, 1993)

data into objects take place around the domain rather than processing. This makes OO systems being more resilient to change as the concept (or data) is less volatile with time. Other claims of OO include increasing productivity, reducing cycle time, reducing the maintenance task and leading to more reliable and efficient systems. All of these are due to the reuse of proven components (i.e. the objects).

While discussing the difference between OO and structured methods, we can find some potential advantages with object-orientation[49]. For example, the paradigm shift leads to economy in applying OO in system building where the designers are 'assembling' a system by prefabricated parts which are stored in libraries. The comparison between modules and objects can also lead to economy as the average size of objects is typically smaller than that of modules. Further we also regard OO as a 'natural' abstraction and as a common 'language' between users and developers. This will enable users to be more active participants in system development. Finally Cook and Daniels (1994) suggest that one important strength of OO is its ability to integrate heterogeneous systems. It can be predicted that in the future the focus on a particular programming language will become less and less when systems are constructed from parts written in different languages, which will communicate via language-independent structures such as CORBA[50].

Object-orientation has attracted more and more attention in the field of computer science and information technology. One claim is that the designer can use objects as a uniform approach throughout the developing process, this makes the transition from OOA to OOD easy and smooth (Morris et al., 1996). That is, it is assumed the objects identified in the problem domain (analysis) have the same meaning in the solution domain (design). However some researchers (e.g. Kaindl, 1999; O'Callaghan, 1994; Morris et al., 1996) argue this it is difficult for OO to move directly from OOA to OOD because their objects represent different things. To summarise these researchers' points, OOA objects are objects in a *problem* domain which are commonly-held abstractions for real-world things; whereas OOD objects are objects in *solution* space which are specialist abstractions concerning a particular domain of

49.Because of the advantage in object-orientation, we can see many conventional languages have started to grow extensions, such as C and C++, Pascal and Object Pascal, etc.

50.OMG's Common Object Request Broker Architecture.

interest. People can also find these two objects have different characters as they are used in totally different models. OOA model is a model for part of problem domain which is an abstraction of real-world things (both existing or non-existing) and developed before the proposed system is built. OOD model is a model of the internal structure of the proposed system which is an abstraction of a solution in terms of objects for building the system. Thus these models have different purposes: OOA model is to help the analyser to understand the domain and capture requirements; whereas OOD is to help designers to understand the nature of proposed systems. Thus the objects used in OOA and OOD are representing different things and this makes the transition from OOA to OOD difficult and problematic.

There is another potential problem with the claim that OO enables the designer to model the real world closely. There are several reasons (some of which are also applied to structured methods) why it is difficult to accommodate real-world features within a model of software:

- The world is a social system, and we cannot simply apply the principles and methods for investigating natural systems to the social phenomena as we are a part of the social system. Because software is a part of the world, the introduction of software will alter the nature of the social system (i.e. will change the 'soft' system boundary).
- The world is dynamic and unpredictable but a model of software is typically static and predictable because its behaviour is preconceived. This static nature of a model of software means that software system "may actually thwart the dynamism of [the world]" (Warboys et al., 1999).
- OO models use message-passing between objects. As Cook (1994) and Cook and Daniels (1994) observe, this is appropriate for describing software execution, but cannot describe what happens in the world.

Some researchers of SSM (e.g. Stowell, 1995b; Checkland, 1993 and 1999; Checkland and Scholes, 1990; Lewis, 1993 and 1995) conclude that this problem stemmed from the ill-defined philosophy of system development – *ontology* vs *epistemology*. That is, the confusion of treating the status of 'entities' or 'objects' as ontological labels for parts of the real world or epistemological devices through which the

nature of the real world may be explored (Lewis, 1995). For epistemological philosophy they recognise the models might be an epistemological device, as Odell (1992) emphasises:

> Object-oriented (OO) analysis should *not* model reality – rather it should model the way reality is understood by people. The understanding and knowledge of people is the essential component in developing systems. (p. 45)

Thus Lewis concludes that OO analysis is still displaying the same conceptual confusions of structured methods and the same lack of concern for the essential philosophy.

2.2.3 The Influence of Object-Orientation on System Development

The introduction of object-orientation does not only change the programming process for system development, but also it influences people's thinking on software modelling and the software process. Fichman and Kemerer (1997) summarise that the influences of OO include new approaches to analysis and design, greater use of iterative and incremental development, and increased emphasis on component reuse. The influence of OO on designed systems (system development) and on human activity systems (i.e. human factors and organisations) is the subject of the next two subsections.

New Culture of System Development

Meyer (1990) argues that object-orientation is not just a programming style but also an implementation of a new view of what the software is, and this view implies a rethinking of the software process. He summarises the comparison between the two cultures of structured and OO methods in Table 2.2. He names conventional culture as *project culture* because its subject is an individual project; whereas the culture of OO as *component culture* as the subject is reusable components.

The reusability of objects can make the development of systems economic, and localise change and minimise effort during the development and maintenance[51]. One important feature of this reusabil-

51.Further using components from the OO library can also guarantee more performance as their behaviour is prescribed and proven once.

ity is the change in the nature of systems. Traditional system development regards the system to be built as an 'end-product' which is based on a preconceived purpose. Developing systems in this way may ignore some potential contribution the systems may be able to make to the whole development infrastructure. In applying OO in reuse, the systems to be developed are required to be adaptive to further requirements, as is appropriate for the evolutionary ('E-type') systems which will be described later in this chapter. The EM approach to development leads to systems with similar properties that have a flexible behaviour to meet the needs of different people and different environments.

	Project Culture	Component Culture
Outcome	Results	Tools, libraries
Economics	Profit	Investment
Unit	Department	Industry
Time	Short-term	Long-term
Goal	Program	System
Bricks	Program elements	Software components
Strategy	Top-down	Bottom-up
Method	Functional, structured, analysis/ design, entity-relation, dataflow, etc.	Object-oriented
Languages	PDL, C, Pascal, etc.	OOPLs such as Java, C++, Smalltalk, etc.

Table 2.2 A Comparison between Project Culture and Component Culture (from Meyer, 1990)

Human Factors and Organisations Affected by Object-Orientation

One of the claims of OO is that it provides a 'common language' for both users and designers. This enables users to have a more active and participatory role in the development, and designers can have a more active attitude to users and response to new requests efficiently due to the use of pre-existing components. The symbols or diagrams used in OO approach (rather than textual specification in conventional ways) are claimed by Sully (1993) to be better graphical metaphors for effective use, since "there will be the fewest possible concepts to juggle with" and this offers 'cognitive economy' for users. Another potential advantage of this is that people may find it is easier to remember class structures (i.e. by the concepts of abstraction and inheritance in OO).

Even though OO has the potential for increasing reuse, many organisations are still failing to take advantage of this. The following are some comments from managers (during interviews by Fichman and Kemerer, 1997):

> "Reuse is good but comes at a cost. People who have sold OO talk about benefits but not the cost".
>
> "The objects that get reused tend to be generic, not domain specific".
>
> "Not many objects are common across systems. When they are, they have different behaviours".
>
> "By the time you define all the behaviours, the business may have changed".
>
> "It sounds like reuse is the right way to go but it does not happen in practice".

Through the interview, Fichman and Kemerer find three reuse barriers: the lack of motive to build for reuse; no processes or tools to manage the reuse environment; and disagreement about the kind of reuse (large or fine grained). In summary, investing in OO for reuse involves overcoming barriers which "potentially put at risk that weight of investment, and only those who are utterly convinced that the weighing of costs and benefits will be in their favour are prepared to take the risk" (O'Callaghan, 1994).

In addition to applying OO in developing software systems, the concept of objects has also been applied to modelling business processes in an organisation. That is, the configurability of objects is used to enable some changes in business processes (i.e. BPR) in order to achieve more productivity for the organisation and produce more value for its customers. For example, the 'business objects' are designed to simulate corporate procedures as well as translate into software objects while implementing the software system to support the business processes. Using OO methodologies can allow for rapid evolution of business objects in response to business conditions. Thus far there have been many researchers focusing on applying OO methods to BPR. For example the work by Jacobson et al. (1995) on BPR (the Object Advantage) is based on their previous software development methods (i.e. OOSE). The use case driven method is firstly applied in requirements gathering and later on applied to model

business processes. Section 3.3 will give details of some researches focusing on linking system development and BPR.

2.3 Circumscription

Circumscription is commonly used in the development of software systems. For example the circumscription of state by the values of a type and of behaviour by the operations of a type or by rules for state change. There is a close link between circumscribing and programming and system boundary. Today it is commonly agreed that the major benefit can be obtained from the application from development activities before coding, such as problems definition, requirements analysis and systems specification (Lehman, 1989). This leads to the development of modelling methodologies which define systematic processes and techniques for formulation of concept and realisation. They are circumscribed because they provide mechanised support for individual activities but also restrict these activities. This section will summarise the relationship between circumscription and programming. Further the role of circumscription and how knowledge is represented and constructed in different paradigms of modelling will also be described.

2.3.1 The Concept of Circumscription

Most of the traditional approaches of software development are trying to formalise the development process abstractly without reference to the particular characteristics of the product to be developed (Sun, 1999). This is mainly because the subjectivity of human beings can influence the process. Also conventionally it was believed that the correctness of a program can be guaranteed as a consequence of a rigorous (mathematical) development process. Thus most methodologies of software development try to minimise the subjective influence of human being in order to ensure the quality of the proposed systems. These prescriptive methodologies tend to specify particular modelling tools and techniques for representing the systems requirements and architectures, as well as set the fixed patterns of activities with rigid algorithms for guiding and controlling the process and thinking of developers. Such circum-

scription is organised according to a particular background and pre-understanding[52]. For example, the structured methods for system development divide the process into several phases in a linear order (both in waterfall model and in spiral model). In each phase the developers have some proven methods, techniques and tools to follow. Pressman (2000) describes this phase-based development process as 'linear thinking'. The aim of such a circumscribed approach is to design high-quality systems with finite resources in a predictable way. We suggest that these prescriptive methodologies focus on only the micro-level aspects, i.e. the constructional aspects of systems such as algorithms or data structures. This differs from the open development and evolutionary paradigm due to Lehman which focuses on the macro-level (or systems level) aspects relating to system-wide attributes and which will be detailed later in this section. Although a prescribed methodology may not provide the deep understanding of the solution domain, it still leads the designers to examine the solution domain more comprehensively and systematically, and to deal in a more uniform way with potential solutions (Beynon et al., 2000c).

Mathematical Models for Circumscribed Behaviour

There are two kinds of systems behaviour: one is defined by the changes of state that are *immediately experienced* and the other is associated with *preconceived* reliable patterns of state change (Beynon, 1994). Traditional mathematical models are concerned with the latter one, where the context for observation is circumscribed and what is to be observed can be anticipated. Since circumscribed behaviour involves some preconception about what is to be observed, it implicitly involves an expression of faith. Beynon infers that we can often use a mathematical model as a basis for description and prediction of a circumscribed behaviour.

As the world is changing and essentially dynamic, its control and exploitation using computing techniques requires the development of restricted representations, made finite, discrete and static through the development of models (Lehman, 1989). Von Bertalanffy (1968) mentions the advantages of math-

52. Winograd and Flores (1987) explain that in most cases this pre-understanding reflects the rationalistic tradition, which includes biases about objectivity, the nature of 'facts' and their origin, and the role of the individual interacting with the computer.

ematical models: unambiguity, possibility of strict deduction, verifiability by observed data. To build a mathematical model for systems behaviour, we create an abstract representation for the observed phenomenon where the effects of state change can be predicted from theory, and for capturing the attributes which are considered relevant and essential to the problems and proposed solutions. The quality of the model depends entirely on how precisely the actual system's behaviour is reflected in the mathematical model (Beynon, 1994). For example we can ask "Does the mathematical model relate the behaviour of a system to its structure?". The way in which a system is developed will affect its response to some exceptional circumstances and failure conditions. But typically an abstract mathematical model does not consider this. Thus the quality of the mathematical model depends on how successfully the system behaviour has been circumscribed. And the process of circumscribing behaviour represents the abstract patterns of human activity which provide the basis for prediction.

Humans versus Computers

The aim of software development is to help people in the manual process of everyday tasks by automatic or semi-automatic systems. But it is commonly the case that computer systems do not provide the same flexibility as manual processes. For example in the late 1950s and early 1960s, many US banks and insurance companies were seeking to use computers to attempt automation, but soon recognised that it was inappropriate to install computers as electronic clerk-replacements (Lehman, 1997). It is important to compare human manual processing of tasks prior to automation with computer processing of the same tasks after the automation, in order to determine the effect of circumscription on system development.

In the manual process (say business process for example), the human judgements are usually influenced by some factors through their exploratory activity. Thus human beings explore some possibilities of different organisations for the resources and make qualitative judgements. The way in which human beings interact with the physical artefacts in the manual process is far less circumscribed than interaction with computers which are programmed in conventional ways. This feature of manual activity is similar to the scientific experiment prior to the identification of the theory because it enables an open-ended

exploration. But this kind of 'opportunistic extension' of functionality may be hindered by the conventionally constructed software systems, "especially where optimised algorithmic processes have been implemented" (Beynon et al., 2000c). Also as such development of computer systems is based on solutions, Lehman (1989) observes that it becomes increasingly difficult or even impossible to return to the previous way of doing business (i.e. the manual process). That is, people are forced by the computer systems to work according to the archetypal plan, and the short-cuts and flexibility in the manual process are no longer possible. This is because the judgements made by humans concern the relationship between the proposed solution (or abstract data) and its referent in the real world, which involve observations and cannot be preconceived and preprogrammed. A rigid algorithm cannot capture all the human awareness of the situation. In this case the developers of software systems need to preconceive any possibilities of integration of the manual and automated activities. As Lehman describes, the knowledge on which their inference of what is required is based represents an historical view.

It is essential to integrate both the manual and the automatic processes in order to exploit the advantages of automation but also retain the features of human manual interaction with physical artefacts. But thus far current systems designed in conventional ways cannot support such integration. Beynon et al. (2000c) argue that the difficulties of integration at the cognitive level stems from the traditional computational paradigms. For example, the difficulty of transition from OOA to OOD stems mainly from the fact that the programming methods rely on circumscription of the application prior to automation. There also exists a distinction between the treatment of observables by humans and the treatment of data by computers, which makes a difference between the conception/construction of software and its execution (Beynon et al., 2002). For the former (software conception/construction), human imagination and knowledge of the real world is essential, whereas for the latter (software execution), human interaction and interpretation is invoked in preconceived ways. Thus it is concluded by Beynon et al. (2002):

> There is a fundamental mismatch between abstract data that is interpreted by the human in direct association with its counterpart in the real-world referent and situation, and abstract data that is manipulated according to computational rules that can only take account of prespecified and preprogrammed features of this association.

We show in this thesis that the situated character of EM approach can allow the modeller to develop artefacts in an open-ended way and without circumscribing the domain.

2.3.2 Open Development versus Closed World

Brödner (1995) reports that in the history of engineering, there are two assumptions on the nature of humans and the function of technology, on the way of seeing the world, and on the human's being in the world:

> One position, ... the 'closed world' paradigm, suggests that all real-world phenomena, the properties and relations of its objects, can ultimately, and at least in principle, be transformed by human cognition into objectified, explicitly stated, propositional knowledge. ... The counterposition, ... the 'open development' paradigm, does not deny the fundamental human ability to form explicit, conceptual and propositional knowledge, but it contests the completeness of this knowledge. In contrast, it assumes the primary existence of practical experience, a body of tacit knowledge grown with a person's acting in the world. This can be transformed into explicit theoretical knowledge under specific circumstances and to a principally limited extent only. ... Human interaction with the environment, thus, unfolds a dialectic of form and process through which practical experience is partly formalised and objectified as language, tools or machines (that is, form) the use of which, in turn, produces new experience (that is, process) as basis for further objectification. (p. 249)

The *closed world* paradigm is characterised by the tradition of rationalism and logical empiricism, which was the predominant way of seeing the world in the enlightenment age and has been the mainstream of Western science and technology (Brödner, 1995). The assumption of this paradigm is that the human world and human work can be entirely modelled as logical propositions and mathematical relations. In particular with the development of computer systems and cognitive science, the closed world paradigm views the world as a 'system' which is controllable, analysable and formally describable. Thus in software engineering, the results of the closed world paradigm are the systems which offer a preconceived framework for interaction between designers, the models and the external world. Traditional [mathemat-

ical] models suit the close-world culture because once they are created, they are independent of its external referents. This means that their relationship to the referents is defined only at the time when the modelling is undertaken, i.e. they represent the modeller's knowledge and understanding at a particular point of time. The knowledge represented in such models is static and circumscribed. Mathematical models are commonly used in the analysis of problems, because computer-based simulation and requirements analysis are based on developing mathematical models and then implementing these as computer programs. The abstract function of the models is based on patterns of interaction which stem mainly from the modeller's understanding of historical results. Whenever the external referents change, the models have to be revised because there is no automatic link between them. To do this may not be easy as a large number of scenarios need to be evaluated.

However we should notice that the continuously changing environment and uncertainty of human behaviour make the predictability and repeatability of the closed world paradigm unrealistic (Winograd and Flores, 1987; Lehman, 1989; Brödner, 1995; Warboys et al., 1999; Sun, 1999; Beynon et al., 2000c). The problem of software development methodologies within this paradigm is that even though they allow for the incremental evolution of requirements, they assume that there is a point in time at which all developers' views will 'converge' and produce a final view of requirements. Paul (1993) terms this the *fixed point theorem*. Such a view neglects the fact that the real-world domain in which the computer systems exist is dynamic and changing. Also in software development, the constructed computer program is related to its external referent only through the relationships which are intended and preconceived by programmers. The computer may terminate operation within some situations which are unintended by the programmers. It is also not clear to what extent we can preconceive which information about the objects is significant in the process of design, or what exceptional scenarios of use need to be considered. Warboys et al. say that it is often the case that the engineer cannot address all aspects of a problem, but will address that part which is capable of being addressed. But because solutions cannot be found for some aspects of the problem does not mean they are unimportant. As they say, "the aspect which is pre-eminent is the fact that owners of systems have in the past been obliged to set down a set of requirements for their systems without much regard for potential problems". Winograd and Flores

comment that this embodies *blindness,* as the essence of intelligence cannot act appropriately when there are some pre-definitions of the problem or the space of states in which to search for solutions. That is, when we are in the domain we have defined, "we are blind to the context from which it was carved and open to the new possibilities it generates". These new possibilities usually create a new openness for design.

With regard to business processes, Warboys et al. point out that "the human issues centre upon the still-developing debate about whether it is useful to prescribe and enforce a detailed prescription of process through the support system". There are many influences on business processes and in many cases it is not possible to define the process which can cope with all possible influences. For instance, the software system is designed to automate manual processes and achieve the goals of the company, but in the closed-world culture the system may become unfit as it cannot be changed as the organisational goals change. The result may be that the users have no flexibility in their operation but are required to work in a way which is defined externally by programmers. "The battery-user has been invented!" (Warboys et al., 1999).

Knowledge Construction versus Knowledge Representation

In general the knowledge manipulation for system development involves two processes (Sun, 1999): *knowledge construction* in which the developer captures the knowledge about the proposed systems, and *knowledge representation* which records the developer's knowledge by means of media such as documents, models or computer artefacts. Conventionally software methodologies focus on knowledge representation and provide the systematic and algorithm-based ways to represent the knowledge in a circumscribed fashion. For example the usual form of knowledge representation in AI is through facts, rules and goals (expressed in propositions). This reflects the closed world paradigm in which formalised mathematical descriptions are used to specify the system behaviour. The knowledge represented in such models or computers is restricted to propositional knowledge in the form of symbolic structures, predicates and rules.

As the models in closed-world culture only represent the developer's understanding at a particular time when the models are created, they are independent of their external referents. Thus if the models fail to reflect the real-world situation, something in the real world domain may be misunderstood. Some researchers argue that the conventionally abstract descriptions which are context-free are inadequate to represent knowledge of a situation fully. Winograd and Flores (1987) say that "knowledge is *always* the result of interpretation, which depends on the entire previous experience of the interpreter and on situatedness in the tradition". A similar emphasis is reflected in EM in the shift of focus from knowledge representation to knowledge construction by which the developer's knowledge is constructed by the context of the real-world domain and in a situated manner (Sun, 1999). This perspective will suit the open development paradigm as the knowledge evolves and is open to change. The focus on knowledge construction does not deny the essential need for knowledge representation: knowledge representation helps the developer to organise the information needed in order to realise a system, whereas knowledge construction enriches the knowledge in a situated manner. For knowledge construction, the knowledge of the developer associated with the system development will change continually due to his growing understanding (or *construal*: the personal understanding of a situation) throughout the process and the new information emerging from the real world domain. It is difficult for conventional models under the closed-world culture to track such change. We shall illustrate in the next chapter how the EM approach takes both knowledge representation and construction into account with the generation and comparison of experience through interaction with computer-based artefacts, but firstly we define the open development paradigm for system development.

Open-Ended Modelling for System Development

The need for interaction between humans and computers makes it impossible to construct a computer model which faithfully represents the real-world situation in the closed-world perspective. As Beynon et al. (1996) argue, whatever formal propositional account is given for the behaviour of the system, *experience* is necessarily involved in the interaction between humans and the system. They point out that the open aspect of computer systems can be reflected in two ways: (1) the user can attribute interpretations

to the system responses which are beyond the scope of what has been formally specified (or circumscribed); (2) the user is enabled to develop skills in interaction with the system which are outside the scope of the abstract system specification. For software development, the software program can be understood as a statement of beliefs about the behaviour which will take place under certain circumstances. However the validity of the beliefs on which the behaviour is founded is always elusive (Warboys et al., 1999). Especially when the software system is employed in an organisation, it requires a precise understanding of the wider implications of the behaviour of the software program with its environment. This is difficult because our environment is complex as well as unpredictably dynamic. When considering the influence of social factors on the system development, Goguen (1994; 1996) concludes the following points:

- Requirements are situated, which result from negotiations whose outcome depends on the participants (their interests, position, background, etc). Furthermore, the construction, interpretation and updating of requirements are also situated as they can only be understood in relation to the concrete situation (i.e. requirements are *local*).
- Requirements are an inextricable part of the development process. That is, requirements are *emergent* – they do not already exist in the minds of users and designers but gradually emerge from the interactions between them. Important requirements issues may arise throughout the lifecycle of system development.
- Requirements are *contingent* – they evolve throughout the development process. Because as development proceeds, new ideas, specification, code, etc are produced which in turn generate new objects and relations and inevitably modify requirements. This is the issue of the 'evolutionary paradigm' for system development which will be detailed in the next subsection.

The *open development* paradigm suggests that the focus should be on interaction that is adaptable in order to cope with the wide variety of situations. That is, systems should not be built by following fixed patterns but be developed by the interaction between actors and the environment. Winograd and Flores (1987) say that the description of programs is based on an *idealisation* in which we preconceive the functionality of computers, but in the actual use of computers there is a critical larger domain in which

new issues arise from the breakdowns of hardware and software. Further, besides these technical aspects, there are the concerns of the people who design, build and use the systems. As Winograd and Flores assert: “The world determines what we can do and what we do determines our world”. The creation of new systems will create new ways which did not exist before and a framework for actions and interactions which may not make sense to people before (cf. section 3.3). Loomes and Nehaniv (2001) observe this situation and describe that the requirements, the system, and even the users may drift, due to the result of situation/interaction change and the pressures within the network of users and systems. They use the word-processing systems as the example to illustrate the drift in requirements and context of use which create new needs. Originally the uses of the first word-processing package was typists and the package was developed as an electronic typewriter. But as new functions and technology are invented and added, such as linking within a network, the requirements of such a system are not merely spelling checks or fonts of variable sizes, but will also include portability, compatability and security. Also, the users of the system will ‘drift’ from typists to the authors or the readers of the document. Through this example, Loomes and Nehaniv conclude that the direction of technological change creates, subverts and co-opts the needs of its users:

> *What the user wants* is now determined by many pressures on the network of organisations that rely on word-processing software, and what the user now *needs* to survive in the organisation. This is how ‘*wants*’ has become ‘*needs*’. (p. 35)

Open development has several advantages. It addresses the important concern that, when interacting with the systems, humans perceive the connection between their actions (or interactions) and the effects of the interaction with the systems. As Brödner (1995) points out, open development means that, rather than restricting the user to specific procedures, the use of a system incorporates some conventions for interaction that enable the user to explore and experience its usability. Open development is also associated with the essential involvement of the designer in interpretative activity associated with interaction with the emerging system during the development process (Beynon et al., 1996). This involvement can be reflected in the development of models and in interaction with the external world.

2.3.3 The Evolutionary Paradigm for System Development

Organisations are human activity systems which are complex as well as dynamic. As mentioned in the last section, it is important for organisations as a whole to be adaptable and flexible, thus the software as a serving system must also be adaptable and flexible. However a software designer cannot predict the role of software systems and how they will contribute to an organisation, especially when such systems will also change the environment. Warboys et al. (1999) infer that software systems should evolve in the way which is consistent with the evolution of the organisation or should even provide a means for the evolution of the organisation. Lehman (1992) also has a similar viewpoint:

> A computer program or software system is a model of a domain and of an application or problem in that domain. For *real* applications in *real* domains, that is for *real world* applications, the domain is essentially continuous, unbounded and dynamic whereas the software model is essentially discrete, bounded and, unless changed by human decision and action, static. Moreover, there will always be a time delay between a decision to change the software and satisfactory completion of the change. The software model is, therefore, an approximation, essentially incomplete and with embedded assumptions. ... With the real world forever changing, an increasing number of assumptions embedded in the system will, unless corrected, become invalid. Even if by some miracle there are none, this can never be known. The system becomes progressively less satisfactory and unpredictable. Uncertainty of behaviour follows.

This means the software development is not a one-off and linear process. It is characterised by uncertainty because we cannot predict the consequences of the new software design. In conventional system development, the software is built and, once installed, it will require only maintenance[53]. Generally the concerns of the maintenance phase are product updates, minor enhancements, error correction and substantial requirement changes (hence rewrites). Warboys et al. (1999) take the view that, while they are in the maintenance phase, software systems should be treated as legacy systems. But Goguen

53.The maintenance phase here is broadly including intensive requirements, design and implementation. But Lehman (1989) suggests the more appropriate term for this phase is *system evolution* as the conventional meaning of maintenance (i.e. preventing change from its current state) is inappropriate when referring to software.

(1994) believes that most of the effort for large systems goes into the maintenance phase. Because not enough effort can be put into being precise in earlier phases and much more is going on during maintenance, i.e. reassessment and re-doing of requirements, design, documentation, testing, etc.

Thus we should treat software as an ‘ever to be adapted *organism*’ but not as a ‘to be produced once *artefact*’ (Lehman, 1989), and new design paradigms should provide useful solutions for the uncertainty situation, for example to design reliable systems which can retain usefulness through the process of adaptation and be able to accommodate the features of human behaviour. Warboys et al. (1999) characterise ‘adaptation’ as the process of implementation of changes to itself which will diminish the difference between a perceived current state and a perceived preferred state. From the systems viewpoint, we need methods for evaluating the systems models by analysing their functionality *during* but not after the development. For the whole system, this goes beyond merely validating and verifying the micro-level aspects such as the data structures or algorithms. We need to develop models in which the artefact can fit the context and satisfy the purpose. This should be done with the wider consideration of the dynamic nature of the contexts in which the systems evolve. Beynon et al. (2000c) add that *compromise* is an important element in human adaptation of solutions: the relaxation of what was considered to be an absolute constraint at the beginning. The software systems and the process of their development must both be adaptable, that is, the design and the system itself should be evolutionary in order to adapt to the environmental changes. The evolutionary paradigm for system development has been considered and discussed by Lehman (1989; 1991; 1997) and Warboys et al. (1999).

The Theory of Evolutionary Design

In (Warboys et al., 1999), after reviewing some software paradigms, such as the Analysis-Synthesis-Evaluation (ASE) paradigm, Artificial Intelligence paradigm, Algorithmic paradigm and Formal Design paradigm, the authors conclude that although each paradigm can address some significant parts of problems, each paradigm is still limited by the ‘bounded rationality’ (the term from Simon, 1996) of the observer (e.g. the designer). That is, the designer cannot be sure of the completeness of the design. This is because the problems in the real world are not usually well-defined, and what we see is condi-

tioned by the events which define the context. Thus sometimes we may ask if the problem in the domain is in fact what it seems to be. According to Warboys et al., problem situations may continue to be poorly defined and therefore might have to be revisited during development until the problem statement and solution are compatible. This is why they characterise the systems as hybrid and what we want is "to discover the information which has lead to the current situation in which we find ourselves and further what information we might need to make the right decision in this situation". They propose the *Evolutionary Design* paradigm which views the software development process as evolutionary:

> A view of the design process where each elaboration of the design is tested against the requirements, and the result of this test determines the form of the next cycle. It recognises that the requirements for the design may need to be changed, so, over the period of design activity, there is continuous testing and mutual adaptation of both design and requirements and they converge to ultimately be consistent. It takes an empirical scientific view of the design activity. (p. 60)

This means the software designers in this paradigm are forced to resort to *satisfactory* rather than *optimal* designs. It is because the solutions, or the final systems, can be influenced by the viewpoint from which the problem is approached, for example the way the requirements are elicited and analysed. There is no right or best solution as the goals of the organisation may change, so the activities would change and new functionalities would be needed for the systems. As the process of software development is regarded as an evolutionary process, a design at any stage is regarded as a tentative solution to the proposed systems[54]. Warboys et al. suggest that this paradigm encourages us to recognise the link between natural and artificial sciences, the use of testable hypotheses as a method, the nature of evolutionary systems and hence the need for an incremental development approach.

54. That is, the *implementation* in any step of the development process becomes the *specification* for the next step. From this perspective, Lehman (1989) determines that the terms 'specification' and 'implementation' reflect points of view but not absolute properties. The only absolute specification is the application itself which undergoes continuing evolution.

The E-Type Systems

Lehman (1989; 1991) identifies three types of software system: S-type, P-type and E-type. An *S-type* program has a well-defined domain which can be represented by a fixed and pre-stated specification. For an S-type program the only criterion of acceptability is its *correctness*, in a strict mathematical sense, for the specification and implementation. Thus program correctness is always regarded as a specific relationship between specification and implementation. A *P-type* program is the one which is required to produce an acceptable solution to solve some stated problems. It is an intermediate classification between S-type and E-type. The criterion of success is that the solution obtained in execution is correct in the sense stated in the problem statement, and the solution may need to be modified if side effects are observed during execution.

An *E-type* program is required to solve a problem in a real-world domain (the ‘E’ stemming from ‘evolution’). The operational domain of the E-type program is unbounded and keeps changing, and the acceptability of the program is determined by its *consequences*. As the E-type program cannot be developed completely and precisely, the criterion of its assessment is the user *satisfaction* with each execution of the system rather than the correctness relative to a fixed specification. Its relationship with the real-world domain is critical and, according to Lehman, the acceptability of the system is dependent on subjective human judgement. It cannot be judged against some separate specification or problem statement but is understood in relation to needs and expectations which arise in the domain of which it is a part (Warboys et al., 1999). Further, as an E-type system is not entirely formal, Lehman describes how validation techniques such as testing or simulation are used to address the *uncertainty* which stems from the non-formal parts of the system. However as the validation techniques are themselves incomplete and imprecise, “the outcome, in the real world, of software system operation is inherently uncertain with the precise area of uncertainty also not knowable” (Lehman, 1991).

Conventional mathematical models which emphasise correctness are suitable for the development of S-type programs. Based on the prescribed and fixed specifications, these models aim to develop the right software (validation) and the software right (verification) (Sun, 1999). Such models provide some

proven techniques and tools for software developing in order to ensure the completeness and accuracy of the software. Lehman does not deny this, but he discusses that all systems relating to the real world are E-type and cannot be entirely formal, even though the end result of the process, the executable code, is totally formal in that it serves as a computer program. This is because the model (or representation) developed in the first step involves a non-formal representation mainly from mental abstraction of referents in the real world, whereas other further steps can produce formal (the S-type) elements. Thus any systems will have to display such characteristics of non-formal parts and omissions of this will be the major source of misuse and unsatisfactory results. The process of E-type system development is a transformation from a non-formal representation of a real-world application to a formal representation of a part of the program. It is possible to develop an S-type program in an E-type context. But the S-type models can only be of limited help for the E-type system. One reason is that their specification can only be a provisional description and is liable to change. Another reason is that the E-type systems have other properties (non-formal parts) which may affect their characteristic emergent properties. Thus Lehman suggests regarding an E-type software as a model of the application, its participants (human and mechanical), the operational domain and activities in that domain. This is similar to Warboys' Theory of Evolutionary Design Paradigm which is claimed to act as a framework able to use other paradigms.

As the E-type software must continually evolve to satisfy the conditions and the requirements of the changing environment, Lehman assesses *feedback* as the most important resource for revising the model because the feedback mechanism plays a significant role in determining the performance and dynamics of the software process (Lehman, 1998; Kahen and Lehman, 2000). Also the feedback nature of software processes has direct implications for the developers of E-type software. In relating to EM, the feedback may take the form of learning from experience by the developer, or take the form of observation or interaction with artefacts. Such feedback can further enable the developer to modify his future behaviour. Lehman says that the software evolution process is a closed loop feedback system. That is, the installation of the software will change the operational domain as well as change the application. Additional activities will also arise from the operation of the systems and there will be changes in the activities associated with the application. Thus the application and the operational domain will be

changed by the output of the process. Another important aspect of feedback in software processes is that it explains why process innovation has only a limited impact at the *global level* (the ‘systems’ level in Boulding’s sense), i.e. the process improvement has to be assessed at the global level. This is because, as Lehman says, the characteristics of systems evolution are largely determined by human and organisational factors, rather than the process and technology used to achieve that evolution. Traditional programming methodologies, which mainly focus on improving the individual development activities and technical processes by proven languages and formalised methods or tools, will not be sufficient. Table 2.3 summarises the behaviour statements (the laws[55]) from Lehman’s observation of the growth of IBM OS/360-370 and other systems, which, as Lehman claims, can provide useful inputs to understanding of the software process.

No.	Brief Name	Law
I (1974)	Continuing Change	E-type systems must be continually adapted else they become progressively less satisfactory in use.
II (1974)	Increasing Complexity	As an E-type system is evolved its complexity increases unless work is done to maintain or reduce it.
III (1974)	Self Regulation	Global E-type system evolution processes are self-regulating.
IV (1978)	Conservation of Organisational Stability	Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an evolving E-type system tends to remain constant over product lifetime.
V (1978)	Conservation of Familiarity	In general, the incremental growth and long term growth rate of E-type systems tend to decline.
VI (1991)	Continuing Growth	The functional capability of E-type systems must be continually increased to maintain user satisfaction over the system lifetime.
VII (1996)	Declining Quality	Unless rigorously adapted to take into account changes in the operational environment, the quality of E-type systems will appear to be declining.
VIII (1996)	Feedback System	E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems.

Table 2.3 Current Statement of the Laws of Software Evolution (from Lehman, 1997; 2000)

55.They use the term *laws* because the observed pattern encapsulate common behaviour that is external to the technical development process (from the developer’s perspective).

2.4 Concluding Remarks

The concept of 'systems' has brought a big change in our view of the world. That is, seeing the world as an integrated whole rather than simply a collection of parts. Systems thinking contains more than holistic thinking as it also emphasises interaction and not just wholeness. Thus systems thinking is also contextual thinking and process thinking. Many concepts in scientific disciplines have been developed from the systems approach. Checkland's SSM based on systems thinking aims to cope with complexity in everyday life. The focus of SSM is on the 'system' which is applied to the process of dealing with the world rather than acting on the world. This makes SSM differ from hard systems thinking which focuses on the 'doing' aspect and on the physical domain. The development of SSM is a learning process and links with both physical and conceptual domains.

The development of computer systems can be of benefit when applying systems concepts to cope with complexity in problems. Conventional systems methodologies in the closed-world paradigm try to formalise the development process based on rigorous mathematical algorithms for minimising the human subjective influence as well as ensuring the quality of the results. The knowledge represented in a circumscribed fashion in such models is restricted to propositional knowledge which is static and circumscribed. What we need is knowledge construction in which the developer's knowledge is constructed in the real-world domain through experience and in a situated manner, i.e. to represent the knowledge in an open-ended way. This meets the culture of open development paradigm and evolutionary design in which the focus is on the interaction between actors and the environment in order to cope with a wider variety of situations rather than following fixed patterns of activities. This is especially important for BPR whose aim is seeking overall business process improvement. We have to consider both the computer supported business processes as well as the development process of the software on which the computer systems are built. The computer system in the domain of BPR is best regarded as an E-type system which needs to be continually adapted to support the organisations with its changing goals, and the changing application in the dynamic environment. For this we need models which can reflect the dynamic properties of the systems, the domains and the interaction between all the agents.

In chapter 4 we will discuss the principles and essential concepts of our experience-based approach – Empirical Modelling. The aim of our EM approach is to allow the results of evolution to feed directly into the decisions about system development. We will show that by EM the evolution and feedback through the interaction with computer-based artefacts can also be useful for human beings to take a global view in the dynamic context.