Chapter 3

Empirical Modelling and Software System Development

Software system development (SSD) is a process whereby human agents – including developers, users and other participants – construct and use a software system for their practices. The basic approach of traditional software engineering (SE) is broadly to formalise the development process abstractly without reference to the particular characteristics of the product to be developed. This promotes a conventional perspective on SSD where there is a clear separation between the abstract development process and the developed product (that is, the software system). To formalise this process, it is divided into several phases to be performed in a linear order[1]. Each phase is characterised by engineering practice, that is, it involves the application of proven methods, techniques and tools in a systematic and cost-effective fashion. Through such a well-defined phase-based process, it is expected that high-quality software can be produced with finite resources and to a predicted schedule. When a system being developed is well defined, its complexity is relatively low, and the overall project and technical risk are reasonably well understood, such a linear process meets this expectation. Unfortunately, these conditions are rarely achieved in the real world, and the worst thing is that changes in the real world,

[1] In a broader sense, a spiral model [Boe88] is also enacted in a linear order.

including people, information, technology and the process itself, are rapid and inevitable. Thus, a key challenge of SSD, especially for ill-defined and volatile software systems, is not to develop high-quality software product in a one-off shot but rather to adapt its process and its products to a rapidly changing environment [cf. Flo87, Fis93].

This thesis challenges the separation between the development process and the developed product associated with the conventional perspective on SSD. It proposes a radically different strategy for SSD in which the product-under-development is concretely represented by interactive artefacts throughout the development process. The presence of this concrete representation of the product-under-development leads to a development process that is no longer abstract, broadly preconceived and product-independent, but is situated, open-ended and product-specific in nature. For this reason, this thesis does not attempt to achieve greater understanding of the development process through formalisation and description in order to gain better control of SSD[2]. Instead, the development process in this thesis is viewed as a collection of situated activities that arise in the construction and use of the required software system in the real world. From this perspective, the investigation of SSD in this thesis is focused on the interactions between human agents, and between human agents and the product-under-development, as represented by interactive artefacts that reflect the evolving software system (cf. [Flo87, Gog97, Leh98]). Although this broader view of SSD is still in its infancy, the discussion about SSD in this chapter and other chapters (Chapter 4, 5 and 7) exhibits its potential to enhance the suitability and adaptability of the development process and the software system to be developed.

[2] Such attempts have led to the investigation of what are currently called *information systems development methodologies* [AF95, HKL95].

3.0 Overview

The purpose of this chapter is to highlight the powerful potential of using EM as an open development model for SSD. As explained in the previous chapter, EM is, in essence, a situated activity: an interactive and situated process without presumed sequential phases and rigid algorithms. In contrast to the 'linear thinking' (in Pressman's term [Pre97]) of the phase-based development process, EM entails 'experimental thinking' whereby iterative experiments are invoked to adapt the system to the changing environment in an interactive and exploratory fashion. This experimental process is suitable for developing open, ill-defined software systems, whilst linear thinking has difficulty in adapting systems to a rapidly changing environment.

Section 3.1 compares EM with a phase-based process model, with particular reference to the enacted process itself and its enactor. This comparison initially focuses on the fundamental principles of both EM and a phase-based process model. On the basis of traditional rationalism, a phase-based model is concerned with setting patterns of activities by applying engineering principles and concepts for guiding and managing the process of SSD. It is assumed that the enacted process can benefit from these formally-defined, well-structured activities, but this process also suffers from the difficulties of adaptation to the rapidly changing real world due to its adherence to these fixed rigid activity patterns. In contrast, EM concentrates on creating and communicating experience for the modeller through the process of constructing a computer-based model for informing the development of the software system [BCSW99]. Because of its situatedness and openness, EM activity is highly adaptable to the real world, though sometimes at the cost of efficiency and effectiveness.

This section also takes the human dimension of enacting a process model into account. Since the subjectivity of a human agent can influence the enacted process to a

significant extent, most models are intended to minimise this influence in order to ensure the expected quality of the developed system. These models stipulate fixed patterns of activities with rigid algorithms that have paramount importance in guiding and controlling the enactor's behaviour. In contrast, EM is dominated by the modeller and his/her experience. No pattern of activities is given for guiding the process of EM, so that the modeller is able to undertake activities in a situated manner in order to cope with diverse situations arising from the process.

Section 3.2 explores the issue of knowledge manipulation in process models for SSD and in EM. Most traditional process models can be seen as emphasising either *knowledge representation* or *knowledge construction*. Process models oriented towards knowledge representation, such as the object-oriented model [Boo94], structured analysis and design [SS95, Pre97] and the entity-relation model [Che76], focus on the process of recording and communicating the knowledge of human agents (including the developer and the user). They seek to capture the knowledge in advance as completely and accurately as possible and to specify the captured knowledge by using context-free abstractions, for example through textual and diagrammatic metaphors. Process models oriented towards knowledge construction, in contrast, do not rely on the completeness and consistency of the knowledge that is represented in advance and separated from the context of the enacted process. These models, such as prototyping [Rei92] and Rapid Application Development (RAD) [Mar91], seek ways to construct knowledge with reference to the context, and hence focus on the process that enriches the knowledge of human agents in a situated manner. Attempting to take both knowledge representation and knowledge construction into account, EM aims to create a computer-based model to represent the modeller's knowledge associated with the intended system and to use situated modelling to enrich the knowledge. Graphical metaphors and dynamic interaction between the modeller and the computer-based model are exploited for the purpose of

knowledge representation. More significantly, EM enables the modeller to facilitate knowledge construction in a situated manner by using the computer-based model as an interactive, open-ended artefact.

EM is in accord with new trends in process models for SSD, such as prototyping [Rei92, Mar91, And94] and scenario-based analysis [DF98, RSB98, SDV96, WPJH98], in seeking to improve the knowledge of the developer (and/or the user) rather than to prescribe the developer's behaviour [DF98]. Such an improvement can only be achieved to a limited extent, however, if the construction of the developer's knowledge and the construction of the represented knowledge in a representational medium are accomplished independently (cf. throwaway prototyping in [And94]). Instead, EM enriches the modeller's knowledge by 'what if' experiments resembling sensitivity analysis in a spreadsheet and at the same time interactively reconstructs the represented knowledge in response to changes in the modeller's knowledge. This enrichment by means of an interactive representational medium (that is, a computer-based model) gives EM the potential to enhance the developer's knowledge to an even higher degree.

Section 3.3 considers the use of EM as an *open development* model (ODM) for SSD. First, three kinds of software classified by M. M. Lehman are discussed: S, E and P-type [Leh94b]. Special attention is given to the E-type software, which is unbounded, ill-defined and liable to change in its operational domain in the real world. Lehman argues that the software itself is a model whose development and maintenance (that is, its *evolution*, to use his term) must be performed through feedback emerging from its operational domain. Unlike Lehman's feedback system, EM creates the software as a computer-based model in which not only feedback but also experience gained through experiments can be used as resources for the evolution of the software. Moreover, with the user of the developed software system in the role of the modeller, and with the aid of definitive programming, EM enables the software system to evolve in its operational

domain. In this way, the problem of tacit knowledge can perhaps be resolved to a significant extent, and the gap between the developer's and the user's views of the system can be greatly narrowed. This section ends with a discussion of the limitations of EM as an ODM for SSD.

3.1 Open Development versus Closed World

SSD is a soft process referring to the entire life cycle of software production and evolution from the initial concept through definition, design, programming, implementation, operation, maintenance and enhancement, to the eventual retirement of the software [STM95]. In order to enact this complicated process, a flexible and practical process model is needed.

As already explained, most process models for SSD are established on the basis of linear algorithms derived from traditional rationalism [WF86]. The difficulties involved in enacting these models in the real world have been identified in [CS90, PR95, Rac97]. In contrast, EM treats the software system being developed as a computer-based model, and thus develops this software system by situated modelling (a form of situated activity, as introduced in Section 2.1). By means of this situated modelling, the knowledge associated with the system is incrementally embedded into the computer-based model through successive experiments and observation. When the modeller is satisfied that the knowledge embedded in the model is well-matched to the real world domain, the software system presented by the computer-based model is exactly the intended system.

It is very difficult to answer the question: is EM a process model? On the one hand, EM meets the demand that a process model can be enacted to construct a software process for SSD [Rol93]. On the other hand, unlike a process model, EM does not circumscribe the structure of the process by using rigorous algorithms. Apart from the principles and concepts described in Chapter 2, no instrument-like guideline is given for

enacting EM. To clarify the difference between EM and most traditional process models, it is helpful to compare their essential foundations. This comparison is unfolded at two different levels: one is concerned with the process *per se* and the other is associated with the modeller, that is, the enactor of the process.

First, a useful starting-point is provided by P. Brödner's observation concerning two cultures in engineering:

> 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. [Brö95]

Although this observation is concerned with the contrast between two cultures and their paradigms in engineering, the distinction may also be applied to models for SSD. The 'closed world' paradigm is characterised by the tradition of rationalism and logical empiricism that can be traced back to Plato. This tradition has been the mainstream of Western science and technology, and has demonstrated its merits in 'hard sciences'[3]. A

[3] T. Winograd and F. Flores define 'hard sciences' as "those that explain the operation of deterministic mechanisms whose principles can be captured in formal systems" [WF86, p. 14]. This thesis also uses this term in the same sense.

major influence on computer research into process modelling was Miller, Galanter and Pribram's famous book, *Plans and the Structure of Behaviour* [MGP60]. The authors examined everyday life and tried to represent it in a formal way. They proposed the concept of a Plan to explain their observation:

> A Plan is any hierarchical process in the organism that can control the order in which a sequence of operations is to be performed [MGP60, p.16].

This definition highlights the view that a Plan for everyday life is a process controlling the behaviour of both the human and the machine. They maintained that the behaviour of the human could be represented in a hierarchical structure as a program in a computer. This understanding obviously corresponds to the theme of the 'closed world' paradigm and has been widely accepted as the rationale of many process models in computer science. Also, it can be found in the simulation of human behaviour in the AI field [Agr95, Dre79, Hau97] and in the modelling of the software development process in the Software Engineering (SE) field [STM95, Som95, Pre97]. These models begin with the interpretation of a Plan as "a relatively fixed repertoire of commonly employed structure of action" [Agr95]. Then, they investigate the possibility of abstracting or formalising a process as a hierarchical plan for guiding the computer and the human as well.

A process that follows a plan in this sense should be predictable and repeatable. That is, by following the same plan, the actions in each process should generally be of the same nature and should lead to a similar final result [Jal97]. However, the continuously changing environment and uncertainty of human agents make the predictability and repeatability of the 'closed world' paradigm untenable[4]. This central problem becomes even more significant as more far-reaching research attempting to automate complicated

[4] The lack of repeatability has been used to criticise the view that SSD is an engineering discipline [DS97, Ste94, XIA98].

processes in the real world is undertaken. As argued by P. Feyerabend in [Fey75], a fixed plan decreases the freedom for taking actions and accordingly blocks the emergence of new concepts. He highlighted the disadvantages through an examination of historical episodes and an abstract analysis of the relations between idea and action [Fey75]:

> ... the principles of critical rationalism ... and the principles of logical empiricism ... give an inadequate account of the past development of science and are liable to hinder science in the future [Fey75, p.179].

> Modern science has developed mathematical structures which exceed anything that has existed so far in coherence and generality. But in order to achieve this miracle all the existing troubles had to be pushed into the *relation* between theory and fact, and had to be cancelled, by *ad hoc* approximations and by other procedures [Fey75, p.64, original emphasis].

Similarly, L. A. Suchman also illustrates the impotence of a Plan in coping with unexpected real-world situations by examining the interaction between the human and a photocopier with embedded instructions. She argues that such an attempt to abstract a process away from the particular environment in which it is situated is of limited applicability in the real world [Suc87]. J. A. Goguen also maintains that "rigidly following a fixed process model can severely limit adaptation" [Gog94]. Similar criticisms can be found in [Agr95, Dre79, Kir91, Rei65, RK95, Tul95, Weg97].

In accordance with the 'open development' paradigm, the above authors all address the inadequacy of setting fixed patterns of activities with rigid algorithms in guiding or even controlling human behaviour in the real world. To overcome this drawback, some researchers argued that the focus of a process model should be on the interaction between the human and the environment rather than on the activities of the human in a particular environment [Agr95, Bro87, Flo95, LR98, RB74, RK95]. They suggest that a process focusing on this kind of interaction should be able to involve as much improvisation as

possible in coping with a wide variety of contingencies. In other words, SSD should not follow fixed activity patterns but instead be freely carried on by the interaction between the enactor and his/her environment[5].

In the same manner, EM rejects fixed activity patterns for SSD and centres on the interaction between the modeller and his/her environment (including the computer-based model and the referent in the real world). In each situation that is encountered, the modeller in EM is empowered to interact with his/her environment in an open-ended manner in order to maintain the virtual correspondence between the computer model and the referent in the real world as shown in Figure 2-1. Through such situated interaction, the computer-based model that is built up in the process of understanding the software system can come to fulfil the functionality of the required software system.

Apart from the openness and situatedness of the process itself, another key factor affecting the process of SSD is its enactor. Most process models pay less attention to this factor[6]. In EM, each state change of the enacted process is due to the invocation of an improvised interaction between the modeller and the computer model (or the referent in the real world) rather than the execution of prescribed activities. In this activity, no situation encountered in the enacted process is predictable in detail. The modeller has to advance the process by means of situated activities that construct the process of SSD. This human-centred concept is in harmony with the increasingly recognised fact that the human being is an important factor leading to the success of SSD [Pre97, LR98, You98].

In fact, as further examination discloses, when a process model is enacted, a complementary process that resides inside the mind of the modeller is simultaneously

[5] The enactor's environment could involve human agents, such as other developers and users. In addition, in a broader sense, both the developer and the user can be an enactor and affect the process of SSD (cf. participatory design in [Mum95]).

[6] Although some models, such as ETHICS [Mum95], have highlighted the importance of participants in the process of SSD, they are more concerned with formalising the process to be followed by participants rather than reflecting what participants do in their practices for SSD.

developed. To clarify this, the former, which changes the state of the environment (including the referent and the computer-based model), is called the *external* process, and

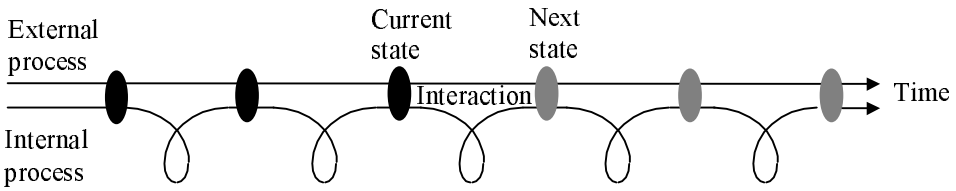

Figure 3-1. The interdependent and inseparable relationship between an internal process and an external process.

the latter, which affects the modeller's knowledge, is called the *internal* process. It is self-evident that both processes are intertwined and inseparable as illustrated in Figure 3-1. This close relationship highlights the fact that the modeller's knowledge (or experience) guides the external process in response to a situation in the real world, and the result from the external process in turn improves the modeller's knowledge through the internal process. Both processes affect each other, and this gives rise to changes in the modeller's mental model, the computer-based model and the referent in the real world.

Hence, the modeller's role in EM is indispensable in construing the phenomena occurring in the referent, in constructing the computer-based model, and, more significantly, in interacting with both the referent and the model in order to maintain their virtual correspondence. It is important to note that the emphasis on the human dimension should not lead to the formalisation of the modeller's behaviour, since this contradicts the principle of open development. Instead, the focus must be on the modeller *per se*. This shift is supported by J. Radford and A. Burton in their comments on simulating human behaviour: "if our aim is to simulate, and thereby gain more insight into, human behaviour, we should begin with the human rather than his behaviour" [RB74, p.349]. That is to say, the primary emphasis should be on the cognitive activity that underlies behaviour rather than on human behaviour itself.

3.2 Knowledge Construction versus Knowledge Representation

SSD is knowledge[7]-intensive [Rob99]. From conceptualisation, description and organisation to transmission, the enaction of a process model is concerned with the manipulation of the knowledge associated with the system being developed. Knowledge manipulation for SSD generally involves two processes: *knowledge construction*, which captures knowledge associated with the system for the developer, and *knowledge representation*, which records the developer's knowledge by means of representational media[8] such as documents and programs. Figure 3-2 depicts the relationship between the developer and these knowledge manipulation processes for SSD. It should be noted that these processes can operate in parallel and without synchronisation.

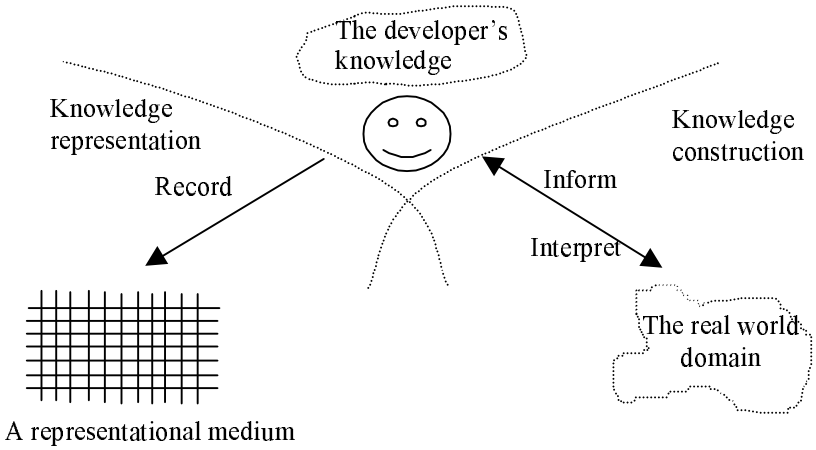

Figure 3-2. Knowledge representation and knowledge construction for the developer

[7] The term 'knowledge' is used loosely to include any structure of information which is constructed by coupling information obtained in one context with other information obtained in a different context [Pre97].

[8] The term 'representational medium' is used as in [Hut95], in a very general way, to indicate any form of media that can be interpreted as a representation of something.

Most traditional process models tend to concentrate on knowledge representation. They take it for granted that the developer can construct knowledge by successfully collecting and interpreting the information in the real world domain. They exploit systematised algorithm-based formats, for example in the form of an entity-relation model [Che76] or object model [Boo94], to represent the constructed knowledge. These representational media, that rely mainly on textual and diagrammatic metaphors[9], can specify the software system and its behaviour in a relatively context-free manner. If the specification fails to reflect the real-world context, this indicates that something in the real world domain is misunderstood. A reinforcing backtrack for locating and correcting errors should then be invoked. In this respect, the representational medium serves as a metaphorical presentation of the real world domain and representation of the modeller's knowledge associated with the system being developed.

However, in practice, there are debates about the completeness and consistency of the knowledge embedded in a representational medium. For example, on the basis of biological experiments, H. R. Maturana argues that symbolic representation cannot serve as the knowledge of an organism to control the way the organism behaves [Mat80]. He claims that an organism can adapt by coupling its structure with its external environment to generate its behaviour. That is to say, the knowledge that controls the behaviour of an organism is context-dependent and open to change. T. Winograd and F. Flores provide a similar argument for the design of an intelligent system that is restricted to representing knowledge by the acquisition and manipulation of the adopted facts. They remark that "knowledge is *always* the result of interpretation, which depends on the entire previous experience of the interpreter and on situatedness in a tradition" [WF86, p.74, original emphasis]. C. Crook also argues that "knowledge is not so neatly circumscribed as to

[9] In practice, metaphors in the form of texts and diagrams have been widely used in computer science for describing and sharing knowledge [Joh94]. Specification is a well-known example.

allow complete and unambiguous stuffing under some human lid" [Cro94, p. 95]. Similar arguments can be found in [Slo90, Cla97, Suc87]. The evidence of these researchers suggest that any attempt to give a full account of knowledge using representational media is inadequate.

Recognising the inadequacy of representing the developer's knowledge by context-free abstractions, some process models, such as prototyping [Rei92, And94] and the spiral model [Boe88] shift their focus from knowledge representation to knowledge construction, where the developer's knowledge is informed by its context in the real world domain. These models indicate that the developer must construct knowledge in a situated manner. From this perspective, knowledge evolves and is open to change. However, specification-based models can only support the openness and evolution of the developer's knowledge to a limited degree. In part, this is because documentation is mainly used for recording and is awkward to change. More importantly, it is difficult to describe context-dependent knowledge adequately by using text-based documentation. Hence, these models tend to contribute to the developer's implicit knowledge, based on practical experience, rather than to an explicit detailed specification. The concept of knowledge construction is consonant with A.diSessa's concept of *knowledge in pieces* (knowledge can only be constructed piece by piece) [diS88], and the theme of constructivism: *knowing-by-doing* (knowledge is gained through practical work) [Puf88].

Knowledge construction is useful for enriching the modeller's knowledge in a situated manner, and knowledge representation is helpful for organising the collected information into relations. Accordingly, EM regards the two approaches as complementary and seeks to take them both into account. To do this, EM uses the computer-based model as an interactive, open-ended artefact both to represent and also to enrich the modeller's knowledge in a significant way. For knowledge representation, it exploits graphical metaphors used in the visualisation of the computer-based model

together with definitive scripts (described in Chapter 2). More importantly, the knowledge represented in the computer model can be explored through interaction with the model. The interactive exploration not only discloses the system's behaviour and the relationship between components, but also enables the modeller to connect knowledge with experience. This practical experience is more useful than the text- and diagram-based metaphors in other models for understanding the represented knowledge in the representational medium (that is, the computer model).

Moreover, the computer-based model in EM is more powerful than the text- and diagram-based metaphors in other models in dealing with changes in the represented knowledge. This is because any change is liable to have implication for the whole system, and the scale of these changes is likely to be reflected in revising the representational media (for instance, in editing a document). However, EM can deal with this problem in a significant way. In EM, any change to the model automatically and interactively leads to a structural change to the model that couples the update (that is, the added definitive script) with the current structure using dependency maintenance, as described in Section 2.4. For example, the structure of observables shown in Figure 3-3(a) is reconstructed to

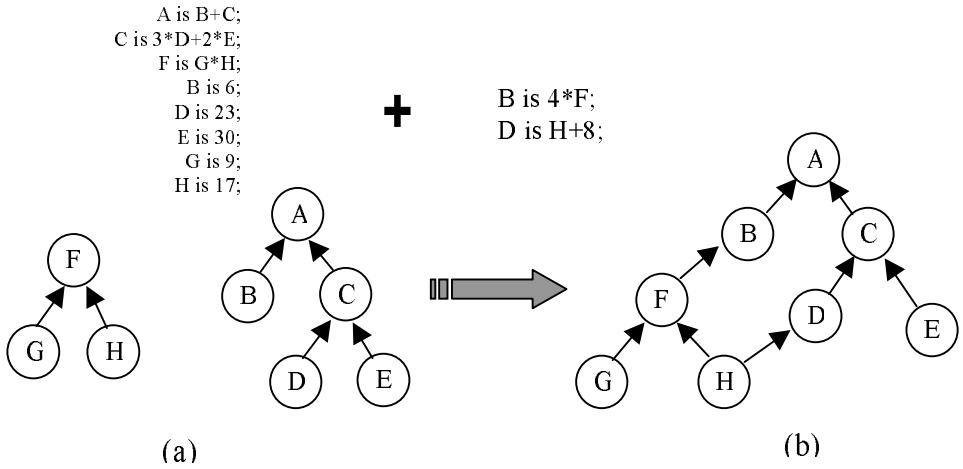

Figure 3-3. The situated structural coupling of observables

Figure 3-3(b) when a new definitive script is added. In other words, given the computer-based model, the modeller can revise the representational medium in an interactive manner. This is very hard to do in most traditional models.

In effect, this structural coupling of the computer-based model is very useful for supporting knowledge construction. For knowledge construction, the modeller's knowledge associated with the developing system continually changes in response to the emerging information from the real world domain. It is very difficult for most traditional representational media, such as documents and prototypes, to keep up with this rapid change (cf. the throwaway prototyping in [And94]). With the aid of structural coupling, EM can to a large extent support the interactive reconstruction of the computer-based model.

In addition, EM aids knowledge construction in a significant way. With reference to the openness and evolution of the modeller's knowledge, the structural coupling in EM, as described above, can provide the modeller with sufficient support. The focus here is on the enrichment of the modeller's knowledge through EM. Through 'what if' experiments resembling sensitivity analysis in a spreadsheet, the modeller can produce sufficient resources for the refinement of his/her knowledge (see Section 2.2.2). In this context of modelling a situation, the relationship between the computer-based model and the modeller in knowledge construction is exceedingly subtle. On the one hand, the modeller creates the computer model to represent his/her knowledge associated with the referent in the real world. On the other hand, corresponding to the interaction with the computer model, the modeller gains additional resources to (re)construct his/her new knowledge. In other words, both the represented knowledge that is embedded in the computer model and the modeller's knowledge are intertwined and complementary.

In practice, a benefit of knowledge construction is that it links the modeller's knowledge to experience rather than simply informs the modeller's knowledge from the

context. This is highlighted in A. diSessa's discussion of science education. He stresses that the use of the computer for improving the student's learning involves building and integrating pieces of knowledge in order to achieve the best connection to experience [diS88]. Obviously, the enrichment of the modeller's knowledge by means of interacting with a computer-based model accords well with this constructivist outlook. It also meets the need, identified by Naur [Nau95], for incorporating experiential knowledge to complement the knowledge that is defined and processed by 'logic and rules'. Similar arguments can be found in [Bur91, Cro94, LW91, Rei97, Sal87].

The enrichment of the developer's knowledge through practical experience has increasingly attracted attention in SSD, as software systems have become bigger and more complicated. Prototyping [Rei92, Mar91, And94] and scenario analysis [DF98, RSB98, SDV96, WPJH98] are very popular illustrations of this theme. However, both approaches, by making use of static rather than interactive representational media, inevitably isolate the represented knowledge from the developer's knowledge. In contrast, EM enables the modeller to interact with the computer-based model in a situated manner. In this way, the computer-based model that serves as an open-ended artefact not only facilitates the construction and integration but also the exploration and extension of the modeller's knowledge. In turn, the enriched modeller's knowledge can to a large extent enhance the knowledge represented by the computer-based model. EM thus supports SSD that is guided by the progressively enriched knowledge of the developer. This meets the need, highlighted by J. Goguen in his discussion of requirements understanding [Gog96], for development techniques that harness rather than reject subjectivity.

3.3 EM as an Open Development Model for SSD

In [Leh94b, Leh97, LR98], M. Lehman identified three types of software system: S-type, E-type and P-type. S-type software has a well-defined domain which can be completely represented by a fixed specification. Correctness is absolutely needed for the specification and its implementation. On the other hand, for an E-type software system, its application domain is, of necessity, bounded to programs, but its operational domain in the real world is unbounded and keeps changing. Such a system cannot be developed completely and precisely, and thus correctness becomes meaningless [Leh94a]. Consequently, the criterion of E-type software acceptability becomes user satisfaction with each execution of this software system rather than absolute correctness to a fixed specification. Other details are summarised in Table 3-1.

S-type	• It is completely defined by a fixed specification. • When revision is required, it is viewed as a new specification, and the resulting program is viewed as a new program. • It needs absolute correctness with respect to the specification.
E-type	• It is a model of the application in its real world domain; as such, the solution system contains a model of itself. • It and its operational domain are conceptually unbounded and continually change. • Its consequences under execution are unpredictable. • Human involvement in the application process and its computerised model excludes precise and complete theories/models of domain and application properties, and makes the change in the process and the model unpredictable. • Its development process is an evolutionary process which requires interactions between many human-populated agencies involving a wide variety of knowledge, understanding, experience and authority.
P-type	• It is used to solve specific problems. • It is intermediate between the S- and E-types.

Table 3-1. The summarised features of S-, E- and P-type software [Leh94b]

Traditional process models, which emphasise correctness, are obviously well suited to developing a S-type software system. According to prescribed, fixed specifications,

these process models seek to develop the right software (validation) and the software right (verification). They provide proven methods, techniques and tools for optimising the process of developing such a system in order to ensure the completeness and accuracy of the developed software. However, for an E-type software system, these models can only provide limited help, since its specification provides only a provisional description and is liable to change. In particular, human involvement makes such change much less predictable.

Hence, Lehman suggests that it is helpful to view an E-type software as "a model of the application, its participants (human and mechanical), the operational domain, and activities in that domain" [Leh98, p.41]. Such a model is incomplete, unbounded and easy to change. There is inevitable and irresistible pressure on this model for change on an increasingly extensive scale; hence, one must regard the evolution of this model (that is, its development and maintenance) as a system in the system-theoretic sense. Lehman argues that the system behaves as a self-stabilising feedback system in which feedback, as the most important resource for evolving the model, is derived from the operational domain of the software in the real world. Through feedback leading to corrective or adaptive changes to the E-type software, one can obtain a degree of intellectual control over the software's evolution, thereby mastering it and achieving its sustained improvement.

From this perspective, it is clear that EM is well-suited to be employed for the evolution of E-type software. For example, EM appreciates that a software system is unbounded and apt to change. No fixed, complete and precise specification is needed to guide the evolution of the software system. Instead, the evolution is driven, controlled and directed by the modeller. Also, any feedback emerging from its operational domain can be captured and incorporated into the software in an interactive, situated manner. In

other words, EM offers the support for change management and human involvement that are important features for the evolution of E-type software highlighted in Table 3-1.

More importantly, EM does not simply *regard* an E-type software system as a model, but *creates* it as a computer-based model. This shift is very powerful in supporting Lehman's concern for evolving a software system as a feedback system. A seed model is created in the computer at the beginning in response to the modeller's initial understanding of the software system being evolved. Like a developer's initial understanding, this computer-based seed model is also incomplete, imprecise and liable to change. When feedback in the form of knowledge emerges from the operational domain of the software system, EM enables the modeller to incorporate the captured knowledge into the computer-based model (that is, the software) by definitive programming and structural coupling described in the last section. In this way, a feedback mechanism is provided by EM in an interactive, situated manner. More significantly, this mechanism leads to the evolution not only of the computer-based model but also of the software that is to be developed, which is presented in this model.

The computer-based model created by EM can also serve as an open-ended artefact for the modeller to facilitate the evolution of the software. Through 'what if' experiments, EM enables the modeller to explore, expand and experience his/her understanding of the software through situated modelling (see Section 2.2). As a result, this improved understanding guides the evolution of the software. It should be noted that a computer-based model is very different in character from a computer program. Whereas the latter fulfils a preconceived and specified function, the former can serve as an open-ended aid to conception and design [BCSW99].

A SSD model for open development, in P. Brödner's sense (see Section 3.1), must be able to tackle the continuous change to the software system in order to interactively manage the knowledge emerging from the process. In this regard, EM has a potential to

cope with continuous change at least as effectively as the feedback system proposed by Lehman.

An open development model (ODM) must also support the interaction between an individual developer or user of the system and his/her external environment, and must allow him/her to first capture the practical experience emerging from the interaction and then embed this experience into the system to inform subsequent interaction. In particular, an ODM should allow the software system's user to guide the evolution of the software in response to his/her practical experience. If it fails to support this user-centred evolution, a model cannot be an ODM. This is because, whilst the developer must struggle to elicit the user's practical experience, the problem of tacit knowledge still exists. Unfortunately, most conventional process models, even the feedback system discussed above, take insufficient account of user-centred evolution. Within these models, the developer is still regarded as the only person who is empowered to shape the software system. This developer-centred bias is evident from the fact that most software systems are provided to the end-user in the form of execution codes. Performance considerations apart, the key reason is to prevent the end-user from modifying the system since it is assumed that the end-user is not competent to do so (more details are given in Section 4.4). On this account, the developer-centred models are too weak to be an ODM for SSD.

In contrast, EM, which regards a software system as a computer-based, open-ended model, is able to support the user-centred evolution by means of definitive programming. Like the modeller, the software's user, if sufficiently qualified, is empowered to embed his/her practical experience into the software system simply by introducing new fragments of definitive script as described in Chapter 2. In this way, the software system is open to change in an interactive, situated manner in response to the captured knowledge emerging from practical experience, even in its operational domain.

In summary: EM treats a software system as a computer-based model that – in the light of definitive programming – can evolve through modelling. Knowledge arising from practical experience, including that generates from the operational domain (through feedback, in Lehman's term) and by 'what if' experiments, is interactively incorporated into the system by situated structural coupling (see Section 3.2). The system evolves with the modeller's understanding, and is always liable to undergo further evolution. For this reason, it is plausible to say that EM serves as an ODM for SSD, in particular for E-type software systems.

Not surprisingly, EM, and its supporting tool tkeden in particular, has its limitations as an ODM for supporting SSD. Some of these limitations have been completely or partially overcome by the author's research, but the others still require further work. These limitations are summarised as follows.

1. EM does not support the interaction between multiple modellers.

In a sense, EM can be viewed as a modelling process for an individual, as described in section 2.2. The modeller is the unique user in the enaction of EM. In the same manner, the tool tkeden, supporting EM, is also developed for individual modelling. However, as highlighted by Lehman, the evolution of E-type software generally requires the interaction between many human agents [Leh98]. This limitation motivates one of the main research tasks in this thesis: to extend the framework of EM to a distributed environment.

2. EM provides no formalised method.

In general, EM is a sort of experience-based modelling technique. In order to free the modeller from rigid algorithms imposed on his/her activities, no formalised method for SSD, apart from some fundamental principles and concepts, is given in EM. On the one hand, the experienced modeller can benefit from the freedom to cope with diverse

situations in practice. On the other hand, the naive modeller is often puzzled by the enaction of EM. This dilemma is concerned with a trade-off between the needs of both kinds of modellers. In fact, according to the discussion earlier (Section 3.1 and 3.2), it is very difficult to say whether or not this is really a limitation for SSD.

3. EM does not support project management and quality control.

The most important purposes for using a phase-based process model are to manage the project of SSD and promote the quality of the developed software system [Blu94a, Gib94, Som95]. EM takes no account of either purpose. In the software industry, this limitation might discourage many practitioners from using EM as a model to develop software systems, especially those software systems which have constraints, e.g. time and budget, even though EM provides the advantages of open development.

4. EM has difficulty in supporting a large-scale project.

An immediate cause of this limitation is the supporting tool tkeden. Since tkeden is the subject of ongoing research, insufficient account has yet been taken of its scaleability, and complex dependency between observables, e.g. higher-order dependency such as is discussed in [GYCBC96], is not supported. It is clear that both problems could be relieved if alternative advanced techniques were exploited. However, such relief would only be partial if the heavy load of modelling for a large-scale project is still attributed to only one modeller. It is clear that distributing the heavy load of modelling to many modellers (see Section 4.1), and improving the modelling technique, for example, by providing reusable definitive scripts (as proposed in Section 5.3), are useful ways to overcome this limitation.

5. Tkeden does not support component reusability.

Since no kind of modularity is enforced on the computer-based model in tkeden, reusable components are difficult to establish. Typically, definitive scripts must be given

piece-by-piece, even though some pieces are very similar. For example, in the case of a hotel booking system (described in section 2.4), each room slot has almost the same description, but to a large extent the modeller cannot reuse a definitive script to generate the needed scripts. This limitation prevents the modeller from structuring the developing system and leads to an increase of program size. Hence, maintaining the developed system becomes much harder. The author's research in this thesis helps to address this limitation.

6. Tkeden does not offer powerful tools to support data manipulation.

The only data structure supported by tkeden is the list. For data-intensive software systems, the weak support for data manipulation leads to complications. For example, in the case of the hotel booking system, the reservation data for each room slot on each day is stored in a list. Any manipulation of these lists requires extra effort from both the modeller and the computer. Without the support of a powerful tool for data manipulation, e.g. through the use of a database, it is not easy to use tkeden for developing data-intensive software systems.

7. Tkeden provides limited support for user interface design.

User interface design has increasingly become one of the most important requirements for supporting SSD. Most modern programming languages, such as VB and Java, take this support for granted. In addition, since the interaction between the modeller and the computer-based model is the most important activity in enacting EM, support for user interface design can provide the modeller with useful benefits. Unfortunately, tkeden can only provide such support to a limited extent.