Chapter 6

SD as Systems Development

Chapters 3, 4 and 5 have shown that the artefacts and actions of EM and PD are different from those of SD. This suggests that EM and PD cannot be used as an approach to developing software in the conventional sense. EM and PD are more appropriately applied to the creative development of innovative systems than the methodical transformation of requirements into software that characterizes SD. One way that EM and PD could be construed as approaches to developing software is if SD could be viewed as systems development.

This chapter considers how EM and PD might be used as an approach to developing software based on a generalization of the notions of computer, program and programming. The conventional view of the computer as an electronic device, or embodiment of a Turing machine, is generalized to *computer as artefact*, program as stored program is generalized to *program as system configuration*, and programming as SD is generalized to *programming as configuring systems* that is essentially the activity of EM and PD. The usefulness of this alternative view is assessed by considering how it addresses the topical issues in SD and requirements engineering.

6.1 Generalizing computers, programs and programming

The concepts of computer, program and programming have precise and unambiguous meaning in SD. The notion of computer in SD is characterized by the electronic computer, based on the von Neumann architecture [Asp90, EE90], and Turing's

model of a universal machine [Tur36, SW88], consisting of an unbounded store and finite control unit capable of performing simple operations. Associated with this notion of computer is the concept of the stored program [Asp90, EE90]: a sequence of symbols that are the same as data but can be interpreted as actions by the computer. Subject to this interpretation, programming is the process of constructing the sequence of actions that is the stored program. Constructing the stored program is typically facilitated by the generation of SD artefacts, such as the structure, behaviour and process models and code written in programming languages.

In Chapter 2 the EM notion of the computer as artefact was introduced. The meaning of the term artefact, as used in this thesis, characterized by the abstract boundary that separates the world into form and context, was introduced at the beginning of Chapter 4. The idea of the *computer as artefact* offers a more general concept of the computer than in SD with the electronic computer and Turing machine being particular computer forms. The goal of both electronic computers and Turing machines - to perform the sequence of actions in store - can be realized by other forms given a suitable interpretation of the terms program and programming. This section considers the appropriateness of interpreting programs as system configurations that store the actions of the system within the arrangement of components. Subject to this interpretation, programming is the process of configuring a computer artefact that typically involves activities like EM and PD.

6.1.1 Computer as artefact

In SD the electronic computer and Turing machine embody the notion of the computer. In the lift project the software developer constructed the structure, behaviour and process models during analysis with the intention of using them to design and implement code to execute on a electronic computer. The electronic computer largely determines the nature of the models, methods and tools of SD.

It is an empirical fact that almost all electronic computers consist of the same essential elements [NS76]:

- the store containing sequences of actions and associated data (programs);
- the processor capable of performing the actions in store;

- the input and output devices that support interaction between the processor and environment.

This architecture corresponds to the Turing machine model that contains the essentials of all computers, in terms of what they can do, though other computers with different memories and operations might carry out the same computations with different requirements of space and time. In particular, the model of a Turing machine contains within it the notions both of what cannot be computed and of universal machines - computers that can do anything that can be done by any other machine [Tur36, SW88].

In EM and PD the electronic computer and its environment are combined within the notion of the computer artefact. The meaning of the term artefact as used in this thesis is given at the beginning of Chapter 4. Central to this is Simon's characterization of artefact in terms of form (inner-environment), context (outer-environment) and purpose (the terms form and context are used by Alexander in a similar characterization [Ale67]):

> An artifact can be thought of as a meeting point - an "interface" in today's terms - between an "inner" environment, the substance and organization of the artifact itself, and an "outer" environment, the surroundings in which it operates. If the inner environment is appropriate to the outer environment, or vice versa, the artifact will serve its intended purpose [Sim81].

Adopting this view, the electronic computer and Turing machine are particular forms of the computer artefact. The form is only half of the computer artefact. The computer artefact includes the form, such as the electronic computer or Turing machine, and its context.

The computer artefact is characterized by the goal shared by all computers, including the electronic computer and Turing machine, to perform the sequence of actions in store. The early electronic computers satisfied this goal in virtual isolation. However, an increasing number of actions performed by electronic computers today require corresponding actions in the environment. In combination the store,

processor and input/output devices provide the physical means for satisfying the goal but the successful operation of the electronic computer depends on appropriate actions in its environment: "the form of an artefact is a collection of natural phenomena capable of attaining the goal in some range of environments and the context determines the conditions for goal attainment" [Sim81].

Whereas SD has traditionally placed emphasis on the form of the electronic computer alone, the computer artefact gives equal importance to form and context. Perhaps the main reason for this emphasis in SD is that few generalizations can be made about the context of electronic computers because of their many and varied applications. However, one can say that in general the context of electronic computers typically consist of two kinds of elements:

- people and
- mechanical (including electromechanical) devices.

In the lift project it was observed that the people surrounding the computer at any time tended to be organized into communities, such as modellers, product designers, software developers, lift users and lift engineers. Communities used the computer for a variety of purposes, for example, the modellers and designers used it for creative modelling and design whereas the software developers used it for analysis. It is clear from the artefacts discussed in Chapter 4 that the computer controller would be situated among mechanical components - doors, motors, pumps, buttons, levers - in the lift system.

Modellers and designers use the notion of the computer as artefact to consider alternative computer forms, or to ignore the computer form altogether, during EM and PD. Simon identifies the two related principles of predictability and generality that result from the characterization of artefact in terms of form and context [Sim81]:

- it is often possible to predict behaviour from knowledge of the goal and context, with only minimal assumptions about the form;
- often quite different forms are capable of accomplishing identical or similar goals in identical or similar contexts.

These principles are utilized in EM: predictability means that modellers can focus attention on modelling the context to predict the behaviour of the computer form; generality means that the modeller can consider alternative computer forms so long as they satisfy the same goal as electronic computers and Turing machines in a given context.

The EM notion of the computer as artefact was first introduced in Chapter 2 when describing the unusual status given to the **tkeden** interpreter. In EM the computer is only significant in so far as it serves as a physical instrument with which the modeller interacts. This is in contrast to the way in which the computer is conventionally regarded in classical computer science as a means of implementing an abstract algorithm or computation. In effect, it is how the user apprehends the computer as a physical object that matters in EM, not the invisible mechanism by which the object is specified [BNR95]. As explained in Chapter 2, the view of computer as artefact is necessary for the 1-agent approach to modelling that is essential to EM.

6.1.2 Program as configuration

Conventionally the program is the sequence of actions stored in an electronic computer or Turing machine. This is reflected in the dictionary definition of the term program: "the sequence of actions to be performed by an electronic computer in dealing with data of a certain kind" (cited in [BR92]). The notion of program and program execution has a precise and unambiguous meaning with respect to the von Neumann architecture of the digital computer, consisting of a store, processor and input/output devices, and abstractions thereof [Asp90]. This section considers the meaning of the term program when "computer artefact" is substituted for "electronic computer" in the above dictionary definition [BR92].

When the computer form is an electronic computer or Turing machine the goal of the computer artefact - to perform the sequence of actions in store - embodies the stored program principle. When the form is an electronic computer the meaning of the terms action and store in the goal are precise and unambiguous:

- the actions of the processor are simple read, write and logical operations;

- the store is a large number of neighbouring locations each holding a binary digit denoting an action or data.

Correspondingly, when the computer form is a Turing machine the actions of the finite control unit are specified in terms of simple operations - read, write and scan operations - on the store. The store is typically construed as an infinite tape divided into squares that each hold symbols from a finite alphabet each denoting an action or data [Tur36, SW88].

It has been shown in previous chapters that the structural and functional aspects of a stored program are represented by the artefacts constructed during SD. The structure of the artefacts typically reflect the structure of the stored program: symbols representing actions organized into sequences. The meaning of the artefacts is given, for the most part, by the semantic relationship between the symbols in the artefacts and actions of the processor in a digital computer. Such artefacts were constructed and used during SD in the lift project:

- C++ programs;
- structure, behaviour and process models;
- structure of the statements of requirements.

Each kind of artefact was associated with a particular purpose in the lift project:

- the principal purpose of the C++ program was to be automatically translated by the electronic computer into a stored program;
- the purpose of the structure, behaviour and process models was to assist the software developer in analyzing the requirements of the lift system and represent the information necessary to design and implement a C++ program;
- the software developer used the logical structure of the statements of requirements to construct the structure, behaviour and process models.

All the above representations were linked together into a closed system by automatic translation and manual methods of analysis.

By considering alternative computer forms, such as a mechanical device, a broader view of the stored program emerges. When the computer form is a mechanical system the goal of the computer artefact - to perform the sequence of actions in store - requires a broader and more common sense interpretation of the terms action and store than are associated with electronic computers and the Turing machine model:

- the actions of the mechanical device are those that are made possible by the configuration of components, such as open, close, set, reset, raise, lower, fill and empty;
- the actions are stored within the configuration of components in the mechanical device.

Based on this interpretation the program can be thought of as the configuration of a system. Designers learn the relationship between the form and structure of mechanical devices through the experience of developing systems.

By virtue of the symmetrical nature of interactions, the structure encodes the *sequence* of actions performed by the system in the same way that the stored program describes the sequence of actions performed by a digital computer. Interaction, by definition, involves the synchronization of two actions (this view is adopted in the process calculi CCS [Mil89] and CSP [Hoa85]). An interaction between the system and its context involves an action of the system synchronizing with an action in the context - there is an essential symmetry between form and context. For example, the lift door opening synchronizes with a user pressing a button. By adopting this view of interaction it follows that it would indeed be possible, in principle, to determine the sequence of actions of a system by decoding its structure. The timing of the actions performed by the system may depend on the context but for every action in the context the system is ready with its counterpart action.

It has been shown in previous chapters that the configuration of mechanical systems, like the lift system, can be represented by the artefacts constructed during EM and PD. The structure of the artefacts typically reflect the system structure: metaphorical representations of components organized in space. The meaning of the

metaphors are implied by their shape and context. Such artefacts were constructed and used during EM and PD in the lift project:

- design sketch;
- LSD specification;
- visualization and animation.

Each kind of artefact was associated with a particular purpose in the lift project:

- the purpose of the sketch was to help the designer create a satisfactory design for the system;
- the purpose of the LSD specification was to help the modeller conceive the system in terms of observables and agency without having to think abstractly in terms of structure and function;
- the purpose of the visualization and animation was to creatively explore the structural and functional features that emerge from analysis of the LSD specification.

The artefacts constructed during the lift project were related by the fact that each tended to be progressively more detailed and more abstract.

There are clearly similarities between the nature of SD artefacts and the artefacts constructed during the later stages of EM and PD. In PD the detailed descriptions of components and drawings showing how components are to be arranged are more formal than the sketches produced during conceptual design. Similarly, the scripts defining the animation in the later stages of EM define the function of a system more precisely than the LSD specification or visualization. The general characteristics of the artefacts constructed later in EM and PD have been identified in previous chapters:

- the artefacts are less creative and more analytical than the earlier artefacts;
- linguistic patternment is more important in the artefacts than in the earlier artefacts;

- the meaning of the artefacts is less situated than the meaning of earlier artefacts.

In this way, the SD artefacts in the lift project can be thought of as being similar in nature to the artefacts constructed later in EM and PD that are more analytical, linguistic and unsituated than the artefacts constructed earlier in EM and PD.

The view of a program as a system configuration addresses the essential difficulties of software - complexity, conformity, changeability and invisibility - as identified in Brooks' influential paper entitled "No Silver Bullet: Essence and Accidents of Software Engineering" [Bro87]:

- the complexity of mechanical systems is easier to comprehend than software;
- interfaces between mechanical systems are less of an issue than between software elements;
- modification is essential to mechanical systems but seen as a problem in software;
- mechanical systems are visualizable whereas software is essentially invisible.

A more in-depth discussion of how the concept of the program as configuration addresses the essential difficulties of software follows in Section 6.2.

6.1.3 Programming as configuring

The conventional notion of programming is associated with the construction of a program stored in a electronic computer. In SD the construction of a stored program is given a precise and unambiguous meaning by the methods and models used by the software developer. This section considers the meaning of the term programming when "computer artefact" is substituted for "electronic computer" and "stored program" is substituted for "system configuration" above.

Programming corresponds to SD when the program is stored in an electronic computer. Programming is traditionally used to describe the activity of coding within SD. However, since the purpose of the preceding stages of SD is to determine what is to be coded, the whole process can be construed as programming. In fact,

the SD methods were motivated by the need to elevate programming from a craft to an engineering discipline [Gib94].

Chapter 2 introduces SD as essentially the process whereby a statement of requirements is transformed into code. Most SD methods share the same sequence of stages as were followed during SD in the lift project:

- analysis of the statement of requirements resulting in the construction of structure, behaviour and process models;
- design of the software components and architectures based on the models constructed during analysis;
- coding of the design elements in a suitable programming language.

Each of these stages is characterized by a method to be followed by the software developer and the artefacts that are constructed by following the method. In combination these methods and artefacts are a system animated through the agency of the software developer. This formal system is linked to the stored program by the compiler or interpreter that automatically translates the code into binary digits stored within the electronic computer.

By considering alternative computer forms, such as a mechanical device. a broader view of programming emerges corresponding to EM and PD. When the program is a system configuration programming corresponds to configuring the system. Configuring a system is traditionally associated with manufacture in PD when components are arranged and connected together. However, since the purpose of the preceding stages of PD is to determine the components, arrangements and connections, the whole process can be construed as configuring the system. The stages that precede manufacture are what make configuring the system an engineering discipline instead of a craft [Fer92]. In this way, EM can also be thought of as configuring systems, although there is no product, because it corresponds to the stages that precede manufacture in PD.

Chapters 4 and 5 have identified similarities between the artefacts and techniques in EM and the stage of conceptual design in PD. The PD stage of conceptual design and the related stage of detail design were introduced in Chapter 2:

- conceptual design involves the creative generation and evaluation of design concepts;
- detail design involves focusing on the details of the design concept determining which subsystems and components will provide the desired functionality.

The conceptual design phase is characterized by the creative use of sketches. The detail design phase is characterized by the use of previous knowledge about components, represented precisely and unambiguously as labeled drawings, tables, specifications, formulae and the like, in order to inform the realization of the design concept. The system is manufactured using the detailed drawings produced at the end of detail design.

This progression in conceptual design from the simple to the more detailed concept of a system corresponds to the process of conceptualization that characterizes EM [Bey97] described in Chapter 2:

- interaction with artefacts: identification of persistent features and contexts;
- practical knowledge: correlation between artefacts, acquisition of skills;
- identification of dependencies and postulation of independent agency;
- identification of generic patterns of interaction and stimulus-response mechanisms;
- non-verbal communication through interaction with similar environment;
- situated use of language;
- identification of common experience and objective knowledge;
- symbolic representation and formal languages: public conventions for interpretation.

These stages represent a progression from a subjective to an objective view of the subject. The early stages correspond to the modeller's view of the subject during 1-agent modelling. In the later stages the modeller develops methods of communication as typified in n-agent modelling. Finally, the model acquires a meaning independent of the subject and modeller (0-agent system).

There are similarities between the process of SD and the later stages of EM and PD. The later stages of PD involve an engineer using detailed descriptions of components to determine what arrangements will provide the desired functionality. Similarly, the later stages of EM involve the modeller writing sections of definitive script in order to generates the desired behaviour. The general characteristics of the later stages of EM and PD have been identified in previous chapters:

- the later stages involve less creative exploration of artefacts and the subject than earlier stages;
- actions in the later stages are more predictable than actions in the earlier stages;
- the later stages follow general techniques unlike the earlier stages that are determined more by the specific situation.

In this way, SD can be thought of as corresponding to the later stages of EM and PD when the subject has been conceived and represented formally.

The earlier stages of EM and PD arguably provide a more appropriate framework for requirements engineering than the traditional view of requirements engineering as an extension to analysis in SD. The traditional view of requirements engineering is of the process that generates the statement of requirements previous to SD [Poh96, Dav93, Hof93]. The need to integrate these two activities has resulted in established SD techniques, such as object-oriented techniques, being adopted in requirements engineering. In contrast, the view of requirements engineering as EM and PD is of a continuous process of conceiving a system that progresses in parallel with evolving artefacts. The similarities between EM and PD and the predictions for new directions in requirements engineering identified by Siddiqi and Shekaran, in particular the importance of context, are discussed in Section 6.5.

The activities of EM and PD are in the same spirit as the ways to attack the essential difficulties of software identified by Brooks in [Bro87]: buy versus build; requirements refinement and rapid prototyping; incremental development; great designers.

- reusing artefacts instead of constructing new ones from scratch reduces the cost of invention and at the same time gives continuity;
- exploring the system as it is conceived in EM allows users to clarify their requirements whilst interacting with an up-to-date prototype;
- conceptualization of a system is essentially an iterative and incremental process;
- good conceptual design requires skilled designers.

A more in-depth discussion of the correspondence between the view of programming as EM and PD and the attacks on the essential difficulties of software follows in Section 6.3.

6.2 Addressing the essential difficulties of software

In his acclaimed paper entitled "No Silver Bullet: Essence and Accidents of Software Engineering" [Bro87] Brooks identifies four essential properties of software - complexity, conformity, changeability and invisibility - that are the root cause of problems in SD. In his recent anniversary edition of "The Mythical Man-Month" [Bro95] Brooks develops his theme of the four essential properties of software - complexity, conformity, changeability and invisibility - by reacting, in particular, to the rebuttal paper by Harel entitled "Biting the Silver Bullet" [Har92]. This section considers how these properties are addressed by the view of a program as system configuration in EM and PD and the associated views of the computer as artefact and programming as configuring systems.

6.2.1 Complexity

Brooks makes the observation that software entities are more complex for their size than perhaps any other human constructs because no two parts are alike (at least above statement level) (Table 6.1). A central principle in SD is that similar objects are represented abstractly by a single class. However, as Brooks points out, in this respect, software differs profoundly from computers, buildings and other

Software ...	Systems ...
has no two parts that are alike	have repeated parts
typically has more states than systems	typically have fewer states than software
complexity increases exponentially	complexity increases linearly
complexity is rampant	complexity is managed

Table 6.1: Contrasting complexity in software and systems.

systems, where repeated elements abound. In EM and PD the modeller and designer tend to represent each instance of a part within a system in order to keep a close correspondence between the subject and its representation.

EM has a principled approach to reducing the number of repeated parts in an LSD specification that is not based on classification as in SD. The approach is directly related to the two complementary principles that constitute concurrent engineering in EM [ABCY94c, ABCY94a, ABCY94b] introduced in Chapter 2:

- specifying agents in LSD by considering them in isolation;
- introducing a context for interaction by animating agents using ADM.

It is the context of agents with the same set of observables, dependencies and protocols that distinguish them from one another. For example, it is the position of a person in a lift system that distinguishes them from other people and the position of a brick in a wall that distinguishes it from other bricks. Consequently a single LSD agent specification can represent similar agents by considering them in isolation. The context of agents is the focus of visualization and animation in EM.

Brooks argues that one of the major problems that faces software developers in SD is in dealing with the enormous number of states that a piece of software can have. This is especially true of concurrent programs in which subprograms are changing states independently. Large numbers of states makes it difficult for software developers to conceive, describe and test software. In EM and PD the modeller and designer is not concerned with program states but with the observed states of the subject [BRY90]. Brooks recognizes that software has orders-of-magnitude more states than computers do. In EM and PD the modeller and designer focuses on

the mechanical and electro-mechanical devices that tend to have fewer states than electronic computers or software.

As was discussed in Chapter 4, all relationships between software parts have to be represented explicitly in SD because of its abstract nature. In contrast, relationships that are implied in the subject are not explicitly represented in the model or sketch in EM and PD due to the direct correspondence between the subject and its representation. Brooks observes that, the software elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly. In EM and PD the complexity of structural and functional relationships are dealt with by the powerful computational framework provided by the ADM and the powerful mental processes of the designer familiar with interaction between components.

Brooks concludes his section on complexity by stating his belief that the complexity of software is an essential property: "descriptions of a software entity that abstract away its complexity often abstract away its essence" [Bro87]. Complexity is also important to the product designer who cannot guarantee the quality or safety of an innovative product based on generalized models alone [Pug91, Pug96, Fer92]. The designer must embrace the unfathomable complexities of nature. Similarly, the complexities of natural phenomena are taken head-on by the modeller in EM.

6.2.2 Conformity

Brooks argues that, although scientists have to face complexity, they have a firm faith that there are unifying principles to be found in nature (Table 6.2). He points out that Einstein argued there must be simple explanations of nature because God is not capricious or arbitrary.

Brooks believes that no such faith of unification and simplicity comforts the software developer. Much of the complexity that he must master during SD is arbitrary, forced by the many human institutions and systems to which his interfaces must conform. He argues that these differ from interface to interface, and from time to time, not because of necessity but only because they were designed by different people, rather than by God.

Software ...	Systems ...
has no principle of unification	has the natural principle of unification
conforms to complex interfaces	have few and simple interfaces
is treated as a second-class-citizen	parts are treated equally

Table 6.2: Contrasting conformity in software and systems.

In EM and PD modellers and designers share a similar motivation as scientists to seek simple explanations of systems. Although EM and PD are not sciences in the traditional sense they do involve understanding the nature of objects and using this knowledge to simulate and build systems. The interface, representing structure and function, is of little importance in this activity. EM and PD (and natural sciences) are more concerned with the form and context in combination than the largely artificial boundary that separates them [Sim81].

As Brooks points out, the problem of unifying system components is often left to the software developer because the software is typically the last part of the system and is considered more flexible than other more concrete parts. In this way software is treated like a second-class-citizen within the system resulting in software complexity that could perhaps be better accommodated within mechanical and electrical components. This is not so in EM and PD; they are not two-tier development processes. In EM and PD modellers and designers develop descriptions of software and hardware in parallel as they search for the most appropriate, and usually simplest, model and design solution.

6.2.3 Changeability

Brooks observes that software is constantly subject to pressures for change (Table 6.3). He concedes that buildings, cars, computers and other systems are also but that these systems are infrequently changed after their initial design; they are superseded by later models that incorporate essential changes in order to meet changes in customer need and technology. Significant changes in the concept of a manufactured automobile are infrequent; changes in the concept of a manufactured computer somewhat less so. In turn, Brooks asserts that changes in both the automobiles and

Software ...	Systems ...
is always under pressure to change	concepts are seldom changed
embodies the system function	function is determined by its structure
change originates externally	change originates from the system context
change is rapid	change is slow

Table 6.3: Contrasting changeability in software and systems.

computers are much less frequent than modifications to installed software.

Brooks points out that an often cited reason for the changeability of software is that the software of a system embodies its function, and that the function is the part of the system that most feels the pressure for change. The function of a system is the result of interaction between components and the nature of the interaction is determined by the details of those components. Thus, small changes to components in a system result in changes to the system function. In EM and PD the focus is on the high-level concept of the system, at least in the early stages, rather than the details of components. So, although the concept of a system changes in EM and PD, it changes slower than representations of the system function or component details.

Brooks identifies two processes that conspire to cause software to be changed:

- as a software product is found to be useful, people discover novel uses for it;
- software is adapted to take advantage of new technology.

Arguably both these phenomena are less to do with the software, or even the computer that executes the software, and more to do with the context of the computer [Sim81]. EM and PD provide a broader conceptual framework than SD allowing modellers and designers to address issues that surround the software, such as usability and technological development.

Brooks concludes that software is embedded in a cultural matrix of applications, users, laws, and machine vehicles, and that their changes inexorably force changes upon the software. With this in mind it would seem that the context of the software and the computer executing the software is the important issue rather than the software itself. EM and PD is suited to understanding software in context

Software ...	Systems ...
is invisible and unvisualizable	are visible and visualizable
is abstract except for representations	are concrete as are its representations
involves reasoning and language	involve common-sense and pictures

Table 6.4: Contrasting invisibility in software and systems.

and using this knowledge in order to develop software.

6.2.4 Invisibility

Brooks argues that software is invisible and unvisualizable and that this is in contrast to systems that are inherently visible and visualizable (Table 6.4). He goes on to say that geometric abstractions used in building systems (as used in EM and PD) are powerful tools: "The floor plan of a building helps both architect and client evaluate spaces, traffic flows, views. Contradictions and omissions become obvious. Scale drawings of mechanical parts and stick-figure models of molecules, although abstractions, serve the same purpose. A geometric reality is captured in geometric abstraction" [Bro87].

Brooks continues by saying that the reality of software is not inherently embedded in space making visualization difficult in SD: "[software] has no ready geometric representation in the way that land has maps, silicon chips have diagrams, computers have connectivity schematics. As soon as we attempt to diagram software we realize that it suggests no particular set of symbols for representation or conventions for organizing symbols" [Bro87]. Languages in EM are developed with a particular mode of observation in mind [ABCY94c]. In this way the definitive languages in EM are not arbitrary but correspond to a way of observing the world. The difference between the languages of EM and the pictorial-language of PD is that definitive languages have an operational interpretation as well as a real-world meaning.

In concluding his section on the invisibility of software Brooks comments on the effect this essential property of software has on mental processes and communication: "In spite of progress in restricting and simplifying the structures of software,

Attack ...	EM ...
uses off-the-shelf software products	reuses existing artefacts
cuts cost of development	speeds modelling by reuse
utilizes spreadsheets and databases	tool is based on spreadsheet principles
combines use and programming	combines use and modelling

Table 6.5: EM themes associated with buy versus build.

they remain inherently unvisualizable, and thus do not permit the mind to use some of its most powerful conceptual tools. This lack not only impedes the process of design within one mind, it severely hinders communication between minds" [Bro87].

6.3 Attacks on the essential difficulties of software

As well as identifying the four essential properties of software in [Bro87], discussed in the previous section, Brooks recommends four promising ways to attack the essential difficulties of software - buy versus build, requirements refinement and rapid prototyping, incremental development and great designers - to be included in future approaches to SD. This section considers how these attacks relate to the view a programming as configuring systems in EM and PD and the associated views of the computer as artefact and program as systems configuration.

6.3.1 Buy versus build

Brooks draws attention to the important development in the software industry of off-the-shelf software tools, environments and modules: "Every day it is becoming easier, as more and more vendors offer more and better software products for a variety of applications, for software developers to buy existing software instead of developing it themselves" [Bro87] (Table 6.5).

As was identified in previous chapters, and observed during the lift project, EM and PD are based on the free-trade of modelling and design elements that saves on resources and provides the building-blocks for models and designs. This trade operates at two levels:

- conceptual elements used in design and modelling;
- physical elements used to realize designs.

This free-trade in PD has been noticed by others: "The limits of design are culture-bound: all successful designs rest solidly on specific precedents. Because inventors and designers nearly always devise new combinations of familiar elements to accomplish novel results, links to known technology are inevitably present. The inevitability of the old in the new is no check on originality however. The possible combinations of known elements is subject to endless variation" [Fer92].

Brooks identifies the spreadsheet and simple database systems as perhaps the most powerful general off-the-shelf tools. He argues that these powerful tools, so obvious in retrospect and yet so late appearing, lend themselves to myriad uses, some quite unorthodox. The main software tool used for EM so far is the **tkeden** interpreter. The **tkeden** interpreter shares the same general principles as the spreadsheet [Bey97] and gives the modeller access to scripts rather like a database [BCY94]. These principles have proven their importance in a variety of EM projects, including the lift, OXO [BJ94] (Chapter 2), sailboat [NBY94] (Appendix B), railway [ABCY94c], classroom simulation [Dav96] and VCCS [BBY92] projects.

Brooks believes that the increase in off-the-shelf software tools, environments and modules will blur the distinction between programming and use: "the single most powerful strategy for many organizations today is to equip the computer-naive intellectual workers who are on the firing line with personal computers and good generalized writing, drawing, file, and spreadsheet programs and then to turn them loose" [Bro87]. The EM **tkeden** interpreter and associated definitive languages, such as DoNaLD, ADM, EDEN and SCOUT, aspire to embody the principles underlying such a collection of tools.

6.3.2 Requirements refinement and rapid prototyping

Brooks agrees with the widely held belief that the hardest single part of developing software is deciding precisely *what* is wanted: "No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the

Attack ...	EM ...
addresses what is to be built	models what is observed
determines what client wants	involves understanding the subject
involves rapid prototyping	uses visualization and animations
helps conceive the system to be built	parallels natural conceptualization

Table 6.6: EM themes associated with requirements and prototyping.

interfaces to people, to machines, and to other software. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

Therefore, the most important function that the software developer performs for the client is the iterative extraction and refinement of the product requirements. For the truth is, the client does not know what he wants. The client usually does not know what questions must be answered, and he has almost never thought of the problem in the detail necessary for specification. Moreover, the dynamics of a system are hard to imagine. So it is important to plan for extensive interaction between the client and software developer during SD" [Bro87] (Table 6.6).

It was suggested in Chapter 3 that EM might be construed as a process that precedes conventional SD corresponding to requirements engineering, a theme that is taken up later in this chapter. EM is an interactive process during which a representation of the subject is constructed by the modeller. From this process emerges a definitive script that specifies the system structure and function precisely and unambiguously. Such a description is used to simulate the system using the **tkeden** interpreter and forms the basis of further analysis using conventional SD techniques.

Brooks identifies the tools and approaches to rapid prototyping as one of the most important and successful attacks on the essence of software. He defines a system prototype as something that simulates the important interfaces and performs the main functions of the intended system, while not necessarily being bound by the same hardware speed, size, or cost constraints: "Prototypes typically perform the mainline tasks of the application, but make no attempt to handle the exceptional

Attack ...	EM ...
associates complexity in nature with software	models natural phenomena
advocates top-down approach	begins with high-level concept
results in early working system	maintains up-to-date working model

Table 6.7: EM themes associated with incremental development.

tasks, respond correctly to invalid inputs, or abort cleanly. The purpose of the prototype is to make real the conceptual structure specified, so that the client can test it for consistency and usability" [Bro87].

The computer model in EM has much in common with the system prototype. It provides some of the behaviour of the subject for the purpose of helping understand the subject. However, the system prototype is typically defined with specific interfaces and functionality in mind whereas in the computer model in EM the interfaces and function are not necessarily preconceived. The modeller is able to step-in as super-agent to resolve problems caused by exceptional tasks or unexpected input and incorporate them into the model on-the-fly.

6.3.3 Incremental development - grow don't build software

Brooks gives an account of the history of SD by associating a metaphor for development with each era:

- writing programs;
- building programs (specifications, assembly of components, scaffolding);
- growing programs.

He mentions that the growing metaphor reflects the development of increasingly complex software: "In nature we find constructs whose complexities thrill us with awe. The brain is intricate beyond mapping, powerful beyond imitation, rich in diversity, self-protecting, and self-renewing. The secret is that it is grown, not built" [Bro87] (Table 6.7).

But surely it is better to build rather than grow if the building blocks are available? In EM and PD a model or sketch is built using existing elements if

they are appropriate. This is much less costly in resources than creating elements from scratch. However, when modelling and design elements are not available they have to be synthesized from the basic general concepts of observables, agents and components. Growing a new concept requires ingenuity resulting in something that has an almost mystical quality with the potential for creative discovery, as discussed in previous chapters.

Brooks recommends adoption of the top-down approach to SD: "Mills [Mil71] first proposed that any software should be grown by incremental development. That is, the system should first be made to run, even if it does nothing useful except call the proper set of dummy subprograms. Then, bit-by-bit, it should be fleshed-out in a step-wise fashion, with the subprograms being fleshed-out in turn. This approach necessitates a top-down approach to design in which each added function and new provision grows out of what is already there" [Bro87].

This top-down approach has its counterpart in EM and PD with the design of a concept followed by the consideration of detail. However, the importance of combining top-down with bottom-up design is identified by Pugh [Pug91, Pug96]. It is no good arriving at a concept that is not cost-effective, or perhaps even impossible, to manufacture. Whilst designing the concept for a system the designer should never lose touch with how the concept is to be realized.

Brooks emphasizes the benefits of having an up-and-running system early on in the SD process by using the top-down approach. In EM the modeller is able to animate high-level concepts of a system. Moreover, the modeller is able to animate low-level elements of a system because of the environment provided by the **tkeden** interpreter. This means that EM can provide the benefits of an up-to-date running program during top-down and bottom-up design.

6.3.4 Great designers

Brooks argues that the central question in how to improve the software art centres on people (Table 6.8). This accords with the importance of the modeller and designer in EM and PD as identified in Chapter 3.

Brooks identifies the importance of methodology but also realizes its limita-

Attack ...	EM ...
centres on people	addresses communication between people
advocates creativity	involves creative exploration
is not method based	has no explicit method
centres on individuals	is essentially a 1-agent activity

Table 6.8: EM themes associated with great designers.

tions (Section 5.4.2): "We can get good designs by following good practices instead of poor ones. Good design practices can be taught. Programmers are among the most intelligent part of the population, so can learn good practice. However, the difference between poor conceptual designs and good ones may lie in the soundness of design method, the difference between good designs and great ones surely does not. Great designs come from great designers. Software construction is a creative process. Sound methodology can empower and liberate the creative mind; it cannot inflame or inspire the drudge" [Bro87].

Brooks points out that "the most exciting breakthroughs have been made by individuals. Although many fine, useful software have been developed by committees and built as part of multipart projects, those software systems that have excited passionate fans are the products of one or a few designing minds, great designers" [Bro87]. There are clearly parallels between individuals designing software and 1-agent modelling in EM.

6.4 Software and SD in the future

In a report entitled "Where is Software Headed?" [Lew95] experts in the field of SD from both academia and industry give their predictions for the future of software and SD. This section considers how the views of computer as artefact, program as system configuration and programming as configuring systems in EM and PD relate to their visions of the future of software and SD.

6.4.1 Networked computing and concurrency

A general prediction is the continued move towards networked computing: computing with applications, data, and processing power all dispersed across a network. The conceptual framework underlying conventional SD is based on abstract models which make it difficult to describe and reason about such systems. Networks evolved not for any theoretical reason but because use centred around a single processor and data store was found to be impractical. In EM the notion of the computer as a system should make it easier to understand networks and distributed computing.

Simon recognizes the practical and empirical nature of the development of the early timesharing networked systems: "The research that was done to design computer timesharing systems is a good example of the study of computer behaviour as an empirical phenomenon. Only fragments of theory were available to guide the design of a time-sharing system or to predict how a system of a specified design would actually behave in an environment of users who placed their several demands upon it. Most actual designs turned out initially to exhibit serious deficiencies, and most predictions of performance were startlingly inaccurate.

Under these circumstances the main route open to the development and improvement of time-sharing systems was to build them and see how the behaved. And this is what was done. Perhaps theory could have anticipated these experiments and made them unnecessary. In fact they didn't, and I don't know anyone intimately acquainted with these exceedingly complex systems who has very specific ideas as to how it might have done so. To understand them, the systems have to be constructed and observed" [Sim81]. Such an approach to development accords with systems development in EM and PD.

Concurrency is closely associated in computing with networked or distributed systems. For a network to work each part must have some independence of operation. So, the prediction of a continued move towards networked computing implies a move towards parallel computing. However, Hill, Larus and Wood [HLW95] point out that the conventional programming model is based on the uniprocessor. They argue for a shared address space model for parallel computation. Central to EM is the notion that variables correspond to actual physical features that all share a

common "address space" within the real-world. Conflicts among changing variables are resolved in EM by understanding the correspondence between variables and observables in the subject. A fuller account of concurrency in EM is given in [BSY88, Nes93, Sla90].

6.4.2 Software agents

A central principle of EM is to establish a close correspondence between the computer system and the system perceived or imagined by the software developer. This is achieved by in EM by describing the system in terms of observables and agents. Inevitably, parts of the system acquire names that corresponded to concepts in EM, such as agent and observable. This accords with the predictions of Vetter [Vet95] of the emergence of software agents: distributed computer programs that are capable of carrying out specialized functions on the behalf of humans such as a "knowbot" which intelligently finds information of interest to users over a collection of heterogeneous networked computers. The development of such agents in conventional SD is difficult because the behaviour of the agents depends to a large degree on their environment, which does not generally suit formalization.

6.4.3 Object standards and technology

Another general prediction is the continuation of the Object concept into the future of SD. Meyer [Mey95] says of Object technology "it is here to stay". Consequently there is pressure to make the Object the standard software entity. However, Laplante [Lap95] questions the suitability of the existing Object concept arguing that it is deeply rooted in concepts that evolved in the 1970s with the revolutionary language CLU and in the theories of information hiding attributed to Parnas. This is not to say that the Object is a bad idea, but that it would be worth reviewing the concept in the context of the needs of today. Pree and Pomberger [PP95] argue that establishing standards in Object technology too early could lead to the perpetuation, instead of the solution, of the software crisis. The EM notion of agent offers an alternative to the existing Object concept.

In his book "Object-oriented Software Construction" [Mey88] Meyer presents

a number of standard principles that are traditionally associated with modularity in object-oriented software development:

- the principle of linguistic modular units states that modules must correspond to syntactic units in the language used;
- the principle of few interfaces states that every module should communicate with as few other modules as possible;
- the principle of small interfaces (weak coupling) states that if any two modules communicate at all they should exchange as little information as possible;
- the principle of explicit interfaces states that whenever two modules A and B communicate, this must be obvious from the text of A or B or both;
- the principle of information hiding states that all information about a module should be private to the module unless it is specifically declared public.

In this sense, a module is a more appropriate representation of a software entity than an entity in the real-world. However, texts on object-oriented approaches to SD, including [SM88, SM92] by Shlaer and Mellor, tend to emphasize the direct correspondence between the concept of Object and objects in the real-world. The principles of modularization are seldom observed of entities within the natural world:

- they do not necessarily correspond to a descriptive statement, such as a definition or specification;
- they are not necessarily restricted by the size or number of interfaces they have to other objects (if they can be considered as having interfaces at all);
- they are not necessarily restricted to when they can act;
- they are not necessarily able to make features private.

This suggests that the association between the principles of modularity and real-world objects is inappropriate.

In addition to standards there seems to be a general consensus as to the direction in which Object technology should develop in the future. These include

network stores of Objects to be used in different applications, and specialized Object (horizontal) as well as the traditional application (vertical) development. Pree and Pomberger [PP95] warn that building such higher level standards on the antiquated Object concept will force software developers to produce unnecessarily complicated and unprofessional solutions for problems that could otherwise be solved more efficiently. The EM notion of agent has been adapted by experts in the search for efficient solutions to problems within their own specialist domains, including engineering, SD and education.

Perhaps the main reason why Objects have remained so popular in SD is because they help with the problem of complexity. Laplante says that SD has always been about finding better mechanisms for abstraction to support greater complexity predicting that this trend is set to continue. However, he believes the trend should move away from the use of Objects. Objects achieve their abstraction by generating complex code and relying on complex tools that depend on high-speed modern computers to hide these inefficiencies. Laplante predicts that abstraction will be achieved in the future by tools which harness complex and powerful mental processes to deal with the problem of complexity. Tools based on graphics and real-time interaction instead of formal languages. This principle of providing essentially simple tools and languages that harness the powerful mental processes of the modeller and designer is central to EM and PD.

6.4.4 Product-oriented development

Yet another general prediction is about the move away from the process-oriented approach of conventional SD towards a product-oriented approach in the future. Weide [Wei95] argues that poor design is a major culprit in the software crisis. Many believe it is poor adherence to established engineering processes that is the problem and that this will be improved through proper management. However, Weide makes the point that this assumes that product quality derives largely from process quality. Processes in mature engineering disciplines are of course very important but they came only after successful design had been repeated and observed. EM and PD supports the product-oriented instead of the process-oriented approach by allowing

experts to apply the design knowledge of their own disciplines in developing software.

6.5 Requirements engineering in the future

In a report entitled "Requirements engineering: the Emerging Wisdom" [SS96] Siddiqi and Shekaran identify the direction in which requirements engineering is heading. They predict that the next wave of requirements techniques and tools will account for the problem and development context, accommodate incompleteness, and recognize the evolutionary nature of requirements engineering. This section considers how the views of computer as artefact, program as system configuration and programming as configuring systems in EM and PD relate to the future of requirements engineering. Siddiqi is director of the Computing Research Centre and professor of Software Engineering at Sheffield Hallam University. He is also a founding member of the IEEE International Conference on Requirements Engineering and a permanent member of its steering committee. Shekaran has led a variety of research and development efforts in requirements engineering as a manager in Microsoft.

6.5.1 Emerging importance of context

The importance of context in requirements engineering is a theme that runs throughout the report by Siddiqi and Shekaran.

Siddiqi and Shekaran point out that increasingly practitioners are realizing the traditional approach to SD analysis, involving the decomposition of the problem into parts and the composition of parts, is not appropriate because the process and parts are situated. They argue that the biggest drawback of this reductionist view of partitioning things into smaller parts is that the context will influence the decomposition.

This limitation of the reductionist view is addressed in the philosophical foundations of EM. Traditional empiricism is essentially reductionist based on the principle that phenomena can be reduced into elements of experience. The philosophical foundations of EM, described in Chapter 2, are embodied in "Radical Em-

piricism." Radical Empiricism is based on the presumption that the world is a whole, or "conjunction", with no natural boundaries dividing it into elements: "Conception disintegrates experience utterly" ([Jam96] p.70), "[it] performs on conjunctive relations the usual rationalistic act of substitution - [taking] them not as they are given in their first intention, as parts constitutive of experience's flow, but only as they appear in retrospect, each fixed as a determinate object of conception, static, therefore, and contained within itself" ([Jam96] p.236). How the world is divided is mostly arbitrary depending on the individual.

Siddiqi and Shekaran introduce the views of Jarke and Pohl [JP94] for whom the juxtaposing of vision and context is at the heart of managing requirements: "[Jarke and Pohl] define requirements engineering as a process of establishing visions in context and proceed to define context in a broader view than is typical for an information-system perspective. Jarke and Pohl partition context into three worlds: subject, usage, and system. The subject represents a part of the outside world in which the system - represented by a structural description - exists to serve some individual or organizational purpose or usage" [SS96].

There are clearly parallels between requirements engineering, in the sense of Jarke and Pohl, EM and PD. At the beginning of this chapter the ideas of form and context were introduced along with the notion of computer as artefact in EM and PD. Jarke and Pohl's subject, usage and system correspond to context, purpose and form introduced earlier in this chapter with respect to the notion of the computer as artefact. The term subject, as used by Jarke and Pohl, has the same meaning as the term used throughout this thesis and defined in Chapter 3, that is, the object or system being modeled, designed or analyzed.

Siddiqi and Shekaran identify that, whereas in the past most researchers have focused on functional (or behavioural) requirements, the recent trend has been to direct attention to nonfunctional requirements issues. This recent development brings requirements engineering more in line with PD in which the designer has to consider nonfunctional requirements, such as size, weight, ergonomics, documentation and aesthetics, during conceptual design. The similarity between PD and EM identified in previous chapters suggests that EM also deals with nonfunctional

requirements.

For some time now, argue Siddiqi and Shekaran, the SD community has realized the need to broaden its view of requirements to consider the context within which the system will function, with conceptual modelling being the first step: "Borgida, Greenspan, and Mylopoulos' work [BGM85] on the use of conceptual modelling as a basis for requirements engineering was a major signpost in directing researchers to this perspective" [SS96]. Conceptual modelling and design are central to EM and PD. Both EM and PD involve the process of identifying a high-level concept of the subject and then progressively filling in the detail. This process is sensitive both to the context of the modeller and designer and to the context of the subject.

Siddiqi and Shekaran draw attention to Jackson's alternative way to look at context [Jac95]: "Jackson faults current SD methods for focusing on the characteristics and structure of the solution rather than the problem. Software, according to Jackson, is the description of some desired machine, and its development involves the construction of that machine. Requirements are about purpose, and the purpose of a machine is found outside the machine itself, in the problem context" [SS96]. Jackson's views correspond to those in EM and PD: software as a machine description corresponds to the view of a program as a system configuration or representation thereof; development of software as machine construction corresponds to the view of programming as configuring the arrangement of components in a system. The link between purpose and communities within the context has already been mentioned previously in this chapter with respect to the view of the computer, program and programming in EM.

Siddiqi and Shekaran conclude their account of views on requirements engineering with perhaps the most radical of all and yet probably the one that has most in common with EM and PD: "Goguen argues that requirements are information, and all information is *situated* and it is the situations that determine the meaning of requirements. Taking context (or situations) into account means paying attention to both social and technical factors. Focusing on technical factors alone fails to uncover elements like tacit knowledge, which cannot be articulated. Therefore, an

effective strategy for requirements engineering has to attempt to reconcile both the technical, context insensitive, and the social, contextually situated factors.

For Goguen ... requirements emerge from the social interactions between the users and analysts. This goes beyond taking multiple viewpoints and attempting to reconcile them because it does not attempt, *a priori*, to construct some abstract representation of the system. Current methods of eliciting tacit information, such as questionnaires, interviews and focus groups are inadequate, as Goguen points out.

Instead, he advocates "ethnomethodology" [Gog96]. In this approach, the analyst gathers information in naturally occurring situations where the participants are engaged in ordinary, everyday activities. Furthermore, the analyst does not impose so-called "objective" preconceived categories to explain what is occurring. Instead, the analyst uses the categories the participants themselves implicitly use to communicate" [SS96].

There are clearly parallels between Goguen's view of requirements engineering [Gog94, Gog96, Gog93] and EM and PD: in EM and PD the system begins as a concept within the mind of a single modeller or designer and then is refined into a detailed description of a physical system that can be understood by many. EM is based on the principle of developing languages, such as DoNaLD, SCOUT and ARCA (Chapter 2), that are appropriate for describing particular domains rather than enforcing the use of a single general-purpose language consisting of preconceived concepts. This suggests that an approach to requirements based on the principles of EM and PD would have much in common with the vision of Goguen.

6.5.2 End of requirements as contract

Siddiqi and Shekaran argue that the view of the requirement as contract is rapidly becoming outdated: "Most requirements engineering work to date has been by organizations concerned with the procurement of large, one-of-a-kind systems. In this context, requirements engineering is often used as a contractual exercise in which the customer and the software developer organizations work to reach agreement on a precise, unambiguous statement of what the software developer would build.

Trends in the last decade - system downsizing, shorter product cycles, the

increasing emphasis on building reusable components and software architectural families, and the use of off-the-shelf or outsourced software - have significantly reduced the percentage of systems that fit this profile. The requirements-as-contract is irrelevant to most software developers today" [SS96].

The artefacts of EM and PD provide an alternative to the outdated precise unambiguous statement of requirements. In Chapter 3 it was argued that the properties of the statement of requirements - familiarity, unambiguity, explicit meaning, completeness, consistency and convergence - make it ideal for communication and providing a basis for analysis, however, the properties discourage the creativity needed for new systems. The creative properties of the EM and PD artefacts - novelty, ambiguity, implicit meaning, emergence, incongruity and divergence - make them ideal for individuals to model and design the systems of today but make them unsuitable as contracts.

6.5.3 Supporting market-driven inventors

Siddiqi and Shekaran identify that the bulk of the software developed today is based on market-driven criteria: "The requirements of market-driven software are typically not elicited from a customer but rather are created by observing problems in specification domains and inventing solutions. Here requirements engineering is often done after a basic solution has been outlined and involves product planning and market analysis. Classical requirements engineering offers very little support for these problems. Only recently have researchers acknowledged their existence" [SS96].

This approach reflects that of EM and PD. The first phase of PD is market analysis during which the product design specification (PDS) is formulated. The PDS acts to constrain the essentially creative phase of conceptual design during which the designer invents a system that satisfies the specification. Similarly, in EM the modeller has an idea of the purpose of a system during modelling. In both EM and PD the model or design of a system is not elicited from somebody else but is instead created by the modeller or designer based on an understanding of the context for the system.

6.5.4 Coping with incompleteness

Siddiqi and Shekaran point out that a complete statements of requirements is a rarely achievable ideal: "One impetus for the switch to the evolutionary development model was the recognition that it was virtually impossible to make all the correct requirements and implementation decisions the first time around. Yet most requirements research agenda continue to emphasize the importance of ensuring completeness in requirements specifications. However, incompleteness in requirements specifications is a simple reality for many practitioners. Goguen echoes this view in his criticism of the prescriptiveness of current methods that insist on complete specifications." [SS96].

The findings of previous chapters accord with this view of Siddiqi and Shekaran. In Chapter 3 it was shown that the statement of requirement is complete in SD. The requirements are necessarily complete with respect to the models of analysis so that the formal artefacts of SD and the associated methods combine to form a closed system with which the software developer can derive code. However, it is inevitable that the requirements will change during the development of software [SB82] making such a system approach inappropriate. In EM and PD creativity replaces methodology and creative artefacts replace analytical ones. There is no need for the requirement of a system to be complete in EM or PD.

Siddiqi and Shekaran identify the real challenge of coping with incompleteness as how to decide what kinds and levels of incompleteness the software developer can live with: "To this end we need techniques and tools to help determine appropriate stopping conditions in the pursuit of complete requirements specifications - enabling such clarifications to be postponed to a later development stage" [SS96]. In EM and PD the modeller and designer can see the level of detail in an artefact thus allowing them to judge when their representation of a system is complete. Visualization of software by viewing it as a system configuration should allow for similar techniques as in EM and PD with the potential for developing automated tools to help in the task.

6.5.5 Integrating design artefacts

Siddiqi and Shekaran point out that software developers need faster ways to conveniently express the problem to be solved and the known constraints on the solution: "Often, getting to [the expression of the problem] fast outweighs the risk of over-constraining the design ... requirements engineering becomes more of a design and integration exercise in this context. We need "wide-spectrum" requirements techniques that can capture and manipulate design-level artefacts, such as off-the-shelf components" [SS96].

This corresponds to the generative phase of EM and PD described in Chapter 5 in which the modeller and designer bring together existing artefacts and synthesize new artefacts. Previous chapters have identified the inherent continuity in EM and PD resulting from the reuse of existing artefacts and parts thereof. Reuse means that generation of artefacts is typically done quickly. Tools such as **tkeden** help in this process by allowing artefacts to be combined and animated without restricting the modeller to preconceived combinations of artefact parts.

Siddiqi and Shekaran identify that there have been very few concrete results to date in providing support for the task of evaluating alternative strategies for satisfying requirements. However, they do note the burgeoning interest and activity in requirements tracing may offer some solutions in the near future. The direct correspondence between subject and representation in EM and PD facilitates tracing of artefact features back to features of the subject. After the generation of a sketch in PD it is evaluated against criteria based on the PDS. Similarly, the artefacts generated in EM are explored with respect to the subject.

6.5.6 Making requirements methods and tools more accessible

Siddiqi and Shekaran observe that many practitioners today use general tools like word processors, hypertext links, and spreadsheets for many requirements engineering tasks: "Given the wide variety of contexts in which requirements are determined and systems are built, researchers may be well-advised to focus on specific requirements subproblems and consider building automation support in the form of add-ons to existing general-purpose tools. Less accessible to practitioners are methods that

prescribe a major overhaul of an organization's requirements process and the use of large, monolithic tools" [SS96].

The **tkeden** interpreter in EM embodies the general principles of the spreadsheet. The interpreter improves on the conventional spreadsheet by providing means to define dependencies and the metaphorical representation of variable values in scripts. The modeller is free to extend the basic interpreter by adding more scripts that define underlying algebras for representing the subject within different contexts. The **tkeden** interpreter, and the approach to modelling upon which it is based, has proved accessible to people from various backgrounds. It integrates well into different disciplines, such as engineering and education, and is learned quickly by people in those disciplines. Evidence is in the form of a variety of EM projects in different disciplines, including the lift, OXO [BJ94] (Chapter 2), sailboat [NBY94] (Appendix B), railway [ABCY94c], classroom simulation [Dav96] and VCCS [BBY92] projects.

6.6 Conclusion

This chapter has shown that SD can be viewed as systems development. Central to this is the generalization of the concepts of computer, program and programming in SD:

- computer as artefact;
- program as system configuration;
- programming as the process of configuring systems.

An important result of viewing SD as systems development is that EM can be thought of as an approach to developing software. Evidence in support of EM as an approach to developing software is provided in the way of a favourable assessment of how it addresses topical issues in SD and requirements engineering.

There are those who would argue that there is nothing wrong with the conventional view of SD. After all, there are powerful tools and techniques based on the traditional concepts of computer, program and programming in SD:

- computer as an electronic computer;

- program as a sequence of actions stored in a digital computer;
- programming as the process of constructing the sequence of actions in a digital computer.

There is evidence of these techniques and tools being used successfully in industrial software projects [BH95].

The fact remains that the software industry is in crisis despite the use of powerful methods and automated tools in SD. Reports on the software industry, such as those by Gibbs and Jones [Gib94, Jon95], present a bleak picture:

- for every six new large-scale software systems that are put into operation two others are canceled;
- the average SD project overshoots its schedule by half with larger projects doing even worse;
- three quarters of all large systems are termed operating failures which means that either they do not function as intended or are not used at all.

This crisis is not a recent phenomenon. In the autumn of 1968 the NATO Science Committee convened some fifty top academics and industrialists to discuss the growing problem within the software industry. It was decided during this meeting that SD must be turned into an engineering discipline to solve the software crisis. Gibbs observes that, although this realization was made around a quarter of a century ago, software engineering generally remains a term of aspiration.

It might be argued that at least SD has the essential theoretical foundation required for an engineering discipline whereas EM does not. Shaw, cited in [Gib94], argues that engineering disciplines share common stages of evolution. She has observed parallels between software engineering and chemical engineering. Like software developers, chemical engineers try to develop processes to create safe high quality products as cheaply and quickly as possible. Unlike most programmers, however, chemical engineers rely heavily on scientific theory, mathematical modelling, proven design solutions and rigorous quality control methods.

The state of the software industry nevertheless suggests that perhaps the existing theoretically based computer science is not necessarily the right science for industrial SD. Shaw [Gib94] makes the point that, in comparison with established engineering disciplines, software engineering is less mature. She argues that software engineering is more like a cottage industry than a professional engineering discipline. Although the demand for more sophisticated and reliable software has boosted some large-scale projects to the commercial stage she argues that theoretical computer science has yet to build the experimental foundation on which software engineering must rest. EM provides the flexibility for experimentation and the emergence of theories that are appropriate to particular application domains that is arguably lacking in theoretical computer science.

The conceptual framework of computer science has been extended in the past to address the topics outlined previously in this chapter. Indeed, advances over the years based on additions to the traditional concepts of the computer, program and programming have led to breakthroughs in SD [Bro87]:

- high-level languages;
- object-oriented programming;
- artificial intelligence and expert systems;
- program verification;
- "automatic" programming;
- graphical programming;
- environments and tools.

However, Brooks argues that these advances address the *accidental* difficulties of software: "those difficulties that today attend its production but are not inherent" [Bro87]. Promising attacks on the *essential* difficulties of software have more in spirit with EM than SD, as discussed previously in this chapter. Though it is possible that the current paradigm of theoretical computer science could be extended even

further, Kuhn [Kuh70] warns that over-extending a paradigm eventually leads to its collapse and replacement by another more appropriate set of concepts.

It might be argued that the new concepts of computer, program and programming in EM are too radical and signify too great a departure from the traditional paradigm of computer science. Milner identifies the need for a common framework in which to unite many formalisms: "Computer scientists, as all scientists, seek a common framework in which to link and organize many levels of explanation; moreover, this common framework must be semantic, since our explanations are typically in formal language" [Mil93]. Others call for a complete overhaul of the paradigm of computing: "A new paradigm ... must fundamentally change the way we look at problems we have seen in our past. It must give us a new framework for thinking about problems in the future. It changes our priorities and values, changes our ideas about what to pay attention to and what to consider important" [Lie96].

In conclusion, EM can be thought of as an approach to SD so long as the generalized concepts of computer, program and programming are accepted. As the above arguments and counter-arguments indicate, there seems to be no way of predicting whether the new paradigm will be adopted by the SD community. Kuhn argues that changes from an existing paradigm to a rival paradigm depend on the unfathomable social structure of the community and the social processes by which the community is persuaded to adopt the new paradigm [Kuh70]. But whether or not EM is adopted is surely not as important as the need for the SD community to be actively searching for alternative paradigms in case the existing paradigm does not lead to the all-important science that will form the necessary foundation of software engineering. The existing paradigm might evolve into a paradigm that solves the software crisis, but can the industry afford to wait and see?

6.7 Limitations of EM for developing software

Having concluded in the previous section that EM can be viewed as an approach to developing software it is important to point out a number of practical limitations of EM in this respect. These limitations are characteristic of EM, and do not necessarily apply to approaches to systems development in general. They are consequences

Unsuccessful	Successful
No historical software measurement data	Accurate software measurement
Failure to use automated estimating tools	Early use of estimating tools
Failure to use automated planning tools	Continuous use of planning tools
Failure to monitor progress in milestones	Formal progress reporting
Failure to use design reviews	Formal design reviews
Failure to use code inspections	Formal code inspections
Generalists used for critical tasks	Specialists used for critical tasks
Failure to use formal configuration control	Automated configuration control
User requirements creep > 35%	User requirements creep < 15%

Table 6.9: Patterns of large software systems: failure and success.

of an approach that emphasizes creativity and generality in systems development.

6.7.1 Quality

It is becoming increasingly clear to practitioners that approaches to SD in the future must provide support for quality control [Jon95, Gib94]. It is an empirical fact that testing to find and fix bugs is the most expensive and time-consuming aspect of SD [Jon95, Boe85]. It follows then that the most effective way to reduce the cost and time of software projects is to reduce the number of software defects that reach the test phase of SD. Jones is clear about the importance of quality control: "From a technical point of view, the most common reason for software disasters is poor quality control." Table 6.9 from [Jon95] shows the direct link between successful software projects and the use of defect prevention planning and pretest defect-removal activities.

EM is limited as an approach to SD because it does not provide support for quality control. EM does not provide techniques for dealing with metrics, using estimating and planning tools, monitoring milestones, formally reviewing and inspecting designs, or controlling configurations. Chapter 2 introduces EM as a means by which the modeller represents their conception of the subject as it evolves [Bey97]. Since the techniques used in quality control assume preconceptions about the subject, embodied in methods and automatic tools, it would be unprincipled to

include such techniques within EM except in the most general form.

Pugh's views on quality control in PD provide a useful insight into the issue of quality control in EM and SD. Pugh points out that quality control in PD is traditionally based on mathematics and detailed knowledge about components. This parallels quality control in SD, indicated in Table 6.9 by the use of formal techniques and the associated low requirements creep (less than 15 percent). Pugh argues that the abstract mathematical models and detailed knowledge about components, required to control quality in PD, does not exist in the case of innovative products. In these cases quality can only be specified in general terms, where imposing quality control can have the undesired effect of producing an unsuccessful conventional design instead of a successful innovative one. This accords with the status of quality and its control within EM.

Pugh clarifies his position by contrasting total design with the Quality Function Deployment (QFD) approach to design that, rather like SD, is based on the customer requirement. The difference, as identified by Pugh, is that total design can be performed without a requirement whereas QFD cannot: "QFD evolution is customer requirement/product driven, while the work described in the design core is driven by more fundamental issues, and can be operated in situations where initially there is no product, and hence no 'voice of the customer' " [Pug91]. Pugh sees QFD as becoming increasingly powerful procedure as the design becomes conceptually static. In the same way, the view of EM as an approach to SD is of a process whereby the customer requirement evolves in parallel with the development of the software so that quality control can play an increasingly significant role as development progresses.

6.7.2 Management

It is widely recognized that an improvement in managing software projects has to be made within the software industry. Jones points out that the first six factors in Table 6.9 associated with software disasters are specific failures in the project-management domain, and the next three can be indirectly assigned to poor management practices: "The fact that project-management is the source of so many

problems with software applications means that problems first become visible to customers and upper-management too late for effective damage recovery. Lack of historical measurement of software projects and failure to initially use estimating tools or carefully monitor progress are widespread. This means that projects that get into serious trouble are not identified until very late in development" [Jon95].

EM in this thesis is limited as an approach to SD because it does not provide support for managing the process of development. In other words, EM does not provide support for organizing technical resources and people with the aim of improving the process of development [Pug96]. This thesis has focused on using EM to develop products. By interpreting agents as modellers, designers and software developers EM can be used to model the social and technical context for development. Though this topic is outside the scope of this thesis, concurrent engineering in EM is discussed in these terms in [ABCY94c, ABCY94a, ABCY94b].

Pugh's views on management in PD provide a useful insight into the issue of management in EM and SD. In Pugh's model of design it is assumed that the core phases, as described in Chapter 2, are universal, common to all kinds of design and that it is other areas of design activity that give designs their distinctive character [Pug96]. That is to say, different kinds of design may require different kinds of information, techniques and management. Pugh identifies the area of management as of special importance because design activity requires information, resources, and support to be invested in action in the most effective way. This accords with the view of EM being a general approach to systems modelling that is common to many kinds of development including SD.

Pugh's most recent model, the *business design activity model*, attempts to locate the PD activity firmly within the overall structure of business [Pug96]. The idea is that the design core is constrained not only by the elements of the product design specification - the product design boundary - but also by the elements of the business structure - the business design boundary. If the constraints of the business design boundary are too severe, it will be necessary to take corporate action, restructuring the business to provide designers with more information, more resources, and more support. This notion of management as a context is an appropriate way

to think of the relationship between management and EM.

6.7.3 Methodology

The methodical nature of the SD process is paradoxically both its strength and its weakness. This thesis has shown that the conventional methodical approach to developing software discourages creativity. However, when the requirements for a system are stable the methodical approach can be extremely powerful and successful. This accords with the association, shown in Table 6.9, between a low requirements creep (less than 15 percent), the use of automated tools and formal activities. Perhaps the greatest advantage of a methodical approach is that software developers need only be specialists in the SD method and not in particular real-world domains, such as designing lift systems, sailing, playing OXO and constructing jigsaws.

EM is limited in comparison to SD because it is not a method. EM is not a prescribed sequence of actions to be performed by the modeller. Chapter 2 introduces EM as a means by which the modeller represents their conception of the subject as it evolves [Bey97]. Such a process is necessarily iterative in nature and its details are determined by the complex interactions between the modeller and their environment as they learn about the subject (1-agent modelling). Since methods are reconstructions of previous conceptions of subjects, embodied in general techniques and automatic tools, it would be unprincipled to include methods within the general scheme of EM.

The lack of methodology in EM would probably discourage many practitioners from adopting it as an approach to developing software. However, there are those who believe there has been a general over-emphasis on methodology:

- Kaplan warns that, by pressing methodological norms too far, we may inhibit bold and imaginative adventures of ideas [Kap64] (5.4.2);
- Siddiqi and Shekaran predict a shift away from the requirements as the basis for methodical transformation into code towards creating requirements by observing problems in particular domains and inventing solutions [Sid94];
- Milner argues that the general belief that all systems have to be designed

> within the rich conceptual frame of an existing methodology is wrong and that new methods can be discovered experimentally through building systems [Mil86].

These views accord with the notion that EM can be performed without a method and that methods emerge through doing EM that are specific to particular domains with their own conceptual frameworks.

There are parallels to be drawn between EM and PD with respect to methodology. Part of the success of the Pugh's model of design is that it provides a guide to design rather than prescribes how design should be done: "I regard the model's structure as being analogous to a child's climbing frame: it provides the framework on which to climb, it imparts confidence and safety, yet it doesn't prescribe or predetermine the methods by which the child gets to the top of the frame or indeed around inside it" (Pugh [Pug91] p.50). This accords with the view of EM as a framework for systems development rather than a prescriptive method.

6.7.4 Scale

Future approaches have to scale up to address the problem of SD in large-scale projects. Jones' findings show that most small software projects are successful, but that risks and hazards of cancellation and major delays rise quite rapidly as the application size increases: "the development of large applications in excess of 5,000 function points [or approximately 500,000 source code statements in a procedural programming language] is one of the most hazardous and risky business undertakings in the modern world" [Gib94].

EM is limited as an approach to SD because it does not scale up to large projects. One reason for this limit to scaleability is technical: visualizations and animations have to be simple given the current computer and **tkeden** interpreter technology. Although alternative technologies are being considered it seems that this limitation will always exist if the desired flexibility of the EM tools is to be kept. Another reason for the limit to scaleability is to do with the principles of EM: the modeller must be able to perceive the correspondence between the artefacts and subject. Since this correspondence is central to EM it would be unprincipled to have

artefacts that were incomprehensible because of their size and complexity.

The relationship between EM and SD is similar to the relationship between conceptual and detail design in PD with respect to scaleability. There is clearly a difference between the sketch of a bridge produced during conceptual design and the drawings for the construction of the bridge. The sketch is much simpler and can be easily comprehended by the designer whereas the final drawings are orders of magnitude more complex and typically incomprehensible except by analysis. Moreover, the simple conceptual sketch is essential to the eventual detailed description and construction of the bridge. The process and artefacts of EM can be thought of as the conceptual design and sketch in PD. The method and artefacts of SD can be thought of as the techniques of analysis and the detailed drawing in PD.