3 System Development

Promising applications for EM concepts have been identified in many research areas. These include: software system development [Sun99, Ras01, Maa02], education [Bey97, Roe02], business process reengineering [Eva01, Che02], and product design [Fis01]. These wide ranging potential applications of EM give some empirical evidence of the generality of EM concepts in respect of system development. However, there has been no attempt so far to account for this generality explicitly. In this chapter, we shall explore the generality of EM concepts and argue that the philosophical foundation of EM is domain-independent. In other words, EM concepts can be framed and applied to the development of any kind of system.

The structure of this chapter is as follows. In the first section, we shall review two of the most important research strands in the literature on system development. These strands will be contrasted with the EM perspective on system development discussed in later sections. In addition, we shall specify the aims for the application of EM to system development. In section 3.2, our discussion will focus on thinking about intrinsic properties of systems and their relationship to an EM perspective on system development. In section 3.3, we focus on thinking about system development activities and discuss how EM takes these activities into account. In section 3.4, we conduct a case study of modelling a dishwasher system in both EM and another common modelling approach, object-oriented modelling using UML. Our aim is to explore the differences between these two styles of system modelling. In section 3.5, we review possible ways in which the aims of applying EM to system development set out in section 3.1 can be achieved.

3.1 Research on system development

System development is an area of research that has interested researchers from many different disciplines including engineering, computer science, business studies and philosophy. Despite the general interest, however, there is no agreed definition of

'system'. For example, Jordan [Jor68] lists fifteen different definitions of the term 'system'; the most updated version of Oxford English Dictionary Online [Oed02] also has more than ten different uses of the word. The main reason for this terminological vagueness is that researchers tend to propose their definition of a system as a basis for a prescription for how system development should proceed. This chapter makes no attempt to seek a compromise from the sea of definitions of the term 'system' in the existing literature or to introduce a new one. The purpose of this chapter is to investigate how EM concepts help system development and relate the general principles behind an EM approach to system development to mainstream thinking on system development.

By looking at the literature, we can find two principal perspectives from which system development is discussed. One is generally based on thinking about the nature of systems. The other is based on thinking about activities involved in system development. We shall first discuss the main ideas from both these strands of research which will be contrasted with the EM perspective on system development to be discussed in the rest of this chapter.

3.1.1 System development guided by thinking about systems

Much research on approaches to system development has been guided by thinking about systems. In philosophical terms, the focus is on the ontology and epistemology of systems. Such approaches aim to come up with guidelines about how systems should be studied and developed by first discussing what a system is and how the properties of a system can be determined. The tension between a *reductionist* and a *holistic* view of explaining phenomena in the world has had an important impact on the ways in which system development is conceived.

In the reductionist view, every phenomenon can in principle be explained through an application of a hierarchy of natural laws. Such a reductionist outlook is expressed in the following quotation from Dawkins [Daw86]: "an ecosystem is explained in terms of organisms whose behaviour is explained in terms of proteins and macromolecules and DNA code… until ultimately all behaviour is reduced to The Theory of Everything.". Reductionism presumes that the behaviour of the whole can be understood entirely from the properties of its parts.

The counter to a reductionist view sees the whole as more than the sum of its

parts. Checkland [Che93] justifies this view with reference to a simple example: carbon dioxide cannot be reduced to carbon and oxygen because the properties of carbon dioxide such as inter-atomic distance and bond angles are irreducible. The rejection of reductionism leads to another way of thinking about systems which Capra [Cap96] describes as the holistic view. In the holistic view, a system can only be defined as an integrated whole rather than simply as a collection of parts. The properties of a system *emerge* from interaction between parts that cannot be understood by just studying every part alone [Che93, Cap96].

In practical system development, there is always a tension between these two views – it is not possible to adopt one view and to ignore the other. However, different people might put more emphasis on one view rather than the other. The implications of putting the emphasis on each view can be contrasted as follows:

Abstract representation vs. embodiment. Putting the emphasis on the reductionist view, one tends to believe that knowledge about a system can be reduced to knowledge about components and expressed by using abstract representations such as formal documents. System development is mainly aimed at identifying and building components that constitute the system and creating formal documents that purport to *completely* explain these components and their interaction.

In contrast, when adopting the holistic view, one is more aware of the limitations of formal documents as a representation of knowledge about a system. The emphasis in holistic approaches to system development has been on providing a management structure for synthesising knowledge of the proposed system drawn from a variety of different viewpoints. There are two key principles of this approach to system development: the documents and artefacts generated in the system design embody knowledge about the system that is always open for interpretation; the emergence of the system is associated with management processes for organizing interpretations based on several viewpoints.

Approaches to system development that are holistic in the sense that they take account of many viewpoints on system design may nevertheless adopt representations for systems that are reductionist in spirit (e.g. object-oriented representations). Other approaches to system development, such as the use of genetic algorithms or neural nets, adopt holistic representations. Such approaches can deal effectively with issues of system optimization where the interpretation is constrained, but do not appear to be so well-suited to creative system development where open human mediated

interpretation is required.

Functionality vs. realism. The reductionist view favours the idea that a system can be developed by putting together a prescribed set of functionalities. System development involves identifying and creating the components that can achieve the functionalities necessary to meet the objectives of the system (as e.g. in use-case driven system development [Jac92]). In contrast, a holistic view favours the idea that a system should emerge from modelling – or constructing artifacts within – the projected real world environment for its operation. This is consistent with the philosophy behind alternative object-oriented approaches to system development, which proceed by modelling real world objects without first prescribing the functionalities of the final system.

3.1.2 System development guided by thinking about development activities

In the system development literature, thinking about systems is complemented by thinking about the human activities that are involved in system development, especially those activities that relate to the design of a system. Research interests are along the lines of domain-independent theory of design, empirical studies of design activities, and empirical studies of designers [Log95]. The most relevant research within the scope of our discussion is on the comparison of two paradigms for describing design activities made by Dorst and Dijkhuis [Dor95]: the *rational problem solving paradigm* and the *reflection-in-action paradigm* (cf. Table 2 in Appendix E).

The rational problem solving paradigm originates in theories of rational problem solving advocated by Simon [Sim68]. It sees design as a rational problem solving process. According to Simon, problem solving can be seen as a search process over a solution space. The problem definition is assumed to be stable, and it determines the solution space where a solution lies. In this paradigm, designers are information processors in an objective reality. Here we quote some comments about the rational problem solving paradigm from Dorst and Dijkhuis [Dor95]:

> *"Seeing design as a rational problem solving process means staying within the logic-positivistic framework of science, taking 'classical sciences' like physics as the model for a science of design. There is*

much stress on the rigour of the analysis of design processes, 'objective' observation and direct generalizability of the findings."

It is of interest to note in passing that other researchers, such as Gooding, would take issue with this characterisation of science as rational problem solving (cf. subsection 2.2.3).

The reflection-in-action paradigm advocated by Schön [Sch83] takes a radically different view of the design process. According to Schön, every design problem is unique. The design process is seen as a "reflective conversation with the situation". The designer takes actions according to the current problem situation; these may in turn change the situation on which new actions will be based. In this paradigm, the designer is essentially constructing his or her reality [Dor95].

The implications of adopting these two different paradigms can be contrasted as follows:

Objective observation vs. subjective interpretation. The rational problem solving paradigm places much emphasis on objective observation made by the designer during the design process. This promotes the idea that system development should be done according to well-recognised or standardised methods. In contrast, the reflection-in-action paradigm places much emphasis on the subjective interpretation of the designer. This promotes the idea that system development can be guided by psychological research on cognition.

Generality vs. uniqueness. The rational problem solving paradigm emphasises the importance of identifying the commonality of problems, drawing on abstract knowledge and theories, and reuse of general principles and solutions. System development is typically seen as rationally guided by generic methods. The reflection-in-action paradigm emphasises that every design problem is unique. System development is a negotiation of meaning depending on the designer's situational experiences.

3.1.3 Aims of EM perspective on system development

The EM perspective on system development emphasises capturing the system developers' construal of the emerging system within its environment. EM is not

primarily concerned with modelling the system but modelling for the construal. An EM model for system development can embody a variety of knowledge and experience of the modeller or system developer that cannot otherwise be easily expressed using conventional approaches. EM is in the spirit of constructivist approaches to system development which emphasise “the mental and social nature of the construction of forms of information” [Cro96]. The abstract merits of a constructivist approach to system development are typically discussed without reference to techniques for implementation (cf. [Ras01]). EM provides commonsense concepts and a philosophy of system development that are not only useful for academic discussion of system development but are also helpful in practical system development. Our aims in adopting an EM approach to system development are:

- To promote flexible system design
- To create more reliable systems
- To support the cognitive needs of system development
- To facilitate collaborative work

We shall explain how we believe EM can help us to achieve these aims at the end of this chapter.

In the previous two subsections, we have discussed two ways in which researchers have approached the subject of system development, namely by thinking about the intrinsic properties of a system and by considering the activities involved in system development. These two ways to approach system development are respectively associated with two cultures: research into systems theory, and research into problem solving and design. Both these cultures have an extensive literature but seem to be quite separate.

Modern computer science cannot avoid engaging with both cultures in the construction of large-scale computer-based systems. (e.g. object-oriented analysis(OOA) and design (OOD) is an example of this kind of engagement). Whilst both cultures have quite well-explored foundations, it seems to be hard to put them together except in *ad hoc* ways. The practical consequences are to be seen in the difficulties of integrating OOA and OOD, business process modelling and software development [Fer01], and project management with product design.

In the next two sections, we shall consider how the EM perspective on system development relates to both the research cultures described above (see Figure 3.1). In

section 3.2, we shall explain the EM perspective on system development as it relates to thinking about systems. The discussion will focus on three intrinsic properties of systems: complexity, predictability and unity. In section 3.3, we shall explain the EM perspective on system development as it relates to thinking about activities of system development. The discussion will be focused on three aspects of the activities: cognitive, collaborative and methodological. Explaining the EM perspective from these two directions has two purposes. Firstly, it offers a convenient way to contrast it with other perspectives discussed in the previous two subsections. Secondly, we believe that the picture of EM perspective on system development can only be completed by combining complementary discussions from both research directions.

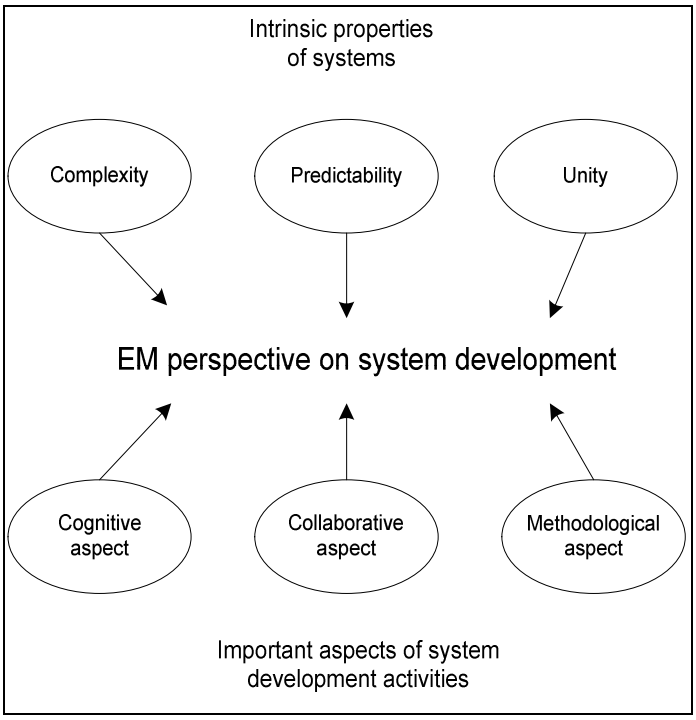


Figure 3.1: EM perspective on system development

3.2 From thinking about systems

Research on systems is generally based on thinking about the parts and the whole of a system. A reductionist view emphasises the importance of parts – it seeks a decomposition of the whole. A holistic view emphasises the importance of wholeness – it is concerned with the emergence of the parts from the whole. All discussions of systems from a traditional perspective centre upon the notions of the parts and the whole.

EM takes a radically different view. We believe that it is more fruitful to take a more relaxed view of part-whole relationships, and think about systems in terms of *observation*. That is to say, we allow the discussion of observation to shape the way we think about systems. Since observation presumes no fixed conception of the part and the whole, this naturally leads to a flexible conception of a system boundary. Systems are the products of observation. Systems emerge from observation. On this basis, we believe that observation is prior to concern for whether a system is best conceived in terms of decomposition or emergence.

Anything that we call a system has three intrinsic properties: complexity, predictability and unity. These properties are intrinsic because we would not call something without all these three properties a system. We shall study each of these in terms of observation along with the implications for system development in relationship to:

- Revealing hidden assumptions
- Exposing complexity in what is superficially simple
- Modelling unreliability
- Modelling for system development that is situated rather than conducted in isolation.

3.2.1 Complexity

Complexity is intrinsic to any system as is especially evident where system development is concerned. This is because we call something a system only when we realise or appreciate its complexity. For example, what we normally call a ‘television’ is a ‘television system’ from the viewpoint of the people who developed it or maintain it. The complexity of a given system or a target system stems from the many viewpoints from which it can be observed. The complexity of a system is reflected in the difficulty of making sense of these viewpoints (e.g. in understanding the relationship between them). Acknowledging that complexity is intrinsic to a system, our real concern in dealing with the complexity of a system in the process of development is with how to manage it and not how to reduce it.

In EM, complexity can be managed by gradually building up a series of observations and representing them within an EM model. Each observation in the EM model reflects what a developer can understand about the system. The observation,

dependency and agency that are embodied in an EM model represent the construal of the system in view of the developer.

It is this construal that manages the complexity and from which the properties and behaviour of the target system eventually emerge. In this respect, EM is more in line with a holistic view than a reductionist view. However, in EM, the system emerges from observations rather than from parts. The distinction between observations and parts is an important one. Studying a system in terms of emergent parts promotes the idea that we can distinguish a specific mode of analysis and a particular viewpoint on the system. Studying a system in terms of emergent observations does not restrict the ways to analyse the system, and therefore encourages conscious management of multiple viewpoints in terms of dependency and agency. EM encourages the evolutionary development of a system with arbitrary complexity from the simplest to the deepest level of understanding. An illustrative example, the noughts and crosses (OXO) model, described in [Gar99] shows this quality. A screen capture of the model is shown in Figure 3.2 below. The purpose of building this model is to study the cognitive processes involved in playing an OXO game. The model was built in an incremental way so that the complexity of each layer of analysis (termed 'cognitive layering') is added as the modeller understands the previous layer. The end result is a comprehensive model that metaphorically represents many aspects of the cognitive processes in playing the OXO game.

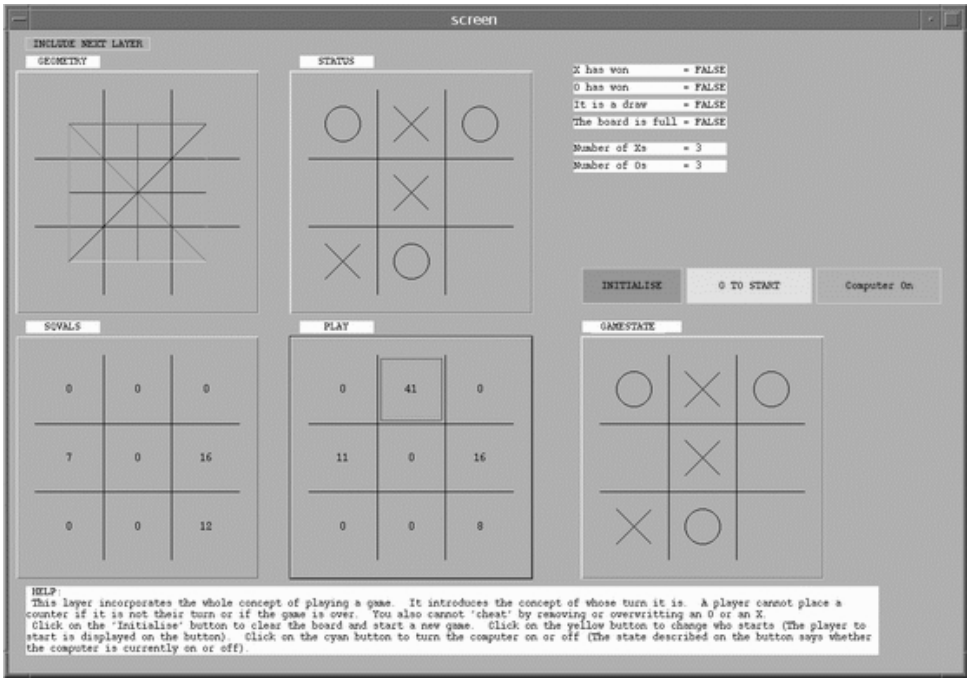


Figure 3.2: The OXO model

3.2.2 Predictability

Predictability is intrinsic to any system because the essence of systematic behaviour is that what happens and can happen is predictable, at least to the extent that the states encountered are preconceived to be possible. The notion of system presumes phenomena (associated with the components of the system and their interaction) together with a system conception on the part of the observer (associated with familiar observations and expectations within a preconceived observational frame). By way of illustration, the *phenomena* that underlie a library system embrace the movement of books into, out of and around the library and the manipulation of catalogues and card indexes. The *observational frame*[1] for such system excludes the height of the chief librarian or the number of steps between floors. *Systematic behaviour* is primarily concerned with the movement of books between identifiable abstract locations such as 'at shelf-mark Q32.4', 'at reception' or 'on loan to borrower Smith'. The *familiar observations and expectations* feature actions such as the transfer of a book from a library assistant to a borrower at the checkout. An action such as handing over the book after first tearing out some pages on the part of the assistant would be outside the scope of preconceived systematic behaviour.

The primary focus in traditional approaches to information system development, such as Jacobson's OOSE [Jac92] is on analysing abstract patterns of interaction that prescribe systematic behaviour without explicit regard for the specific phenomena that will implement them. Critics of this style of OO development argue that the preliminary stages of the system development should involve a more comprehensive analysis of the behaviour of the objects that supports a richer model of the environment in which the system operates. Whichever approach is adopted, the principles used to specify the methods in such objects and to ensure that their combined behaviour is predictable involve an abstraction from situation. This abstraction is ill-suited to relating the system behaviour as conceived to the underlying phenomena involved in realising the system. In EM, a close relationship between the current state of an EM model and the situation it represents is maintained through conducting 'what-if' interactions and recognising patterns of dependency between observables that are characteristic of the system environment.

[1] Observational frame is the context of a system. Put informally, it is the 'things one would expect' from the system.

System predictability is closely related to system reliability. Especially in the context of system development, creating reliable behaviour is the central concern. Reliability is a measure of the exactness of the match between the intended system behaviour and the system's behaviour in use. There are two views of reliability:

- Reliability as a matter of endurance – In this context, our concern is with expected failures. For example, we need to know the extreme values of strength and stress for a car's wheels and test them at the margins. In such testing, the focus is on dealing with the failures that would be expected under extreme conditions. Reliability of this nature can be quantified by gathering measurements and statistics about failures. In this aspect, system reliability can be assessed and predicted by mathematical formulation.

- Reliability as a matter of accident – In this context, our concern is with unexpected failures. The focus is on exposing the hidden assumptions and parameters that lead to unexpected system behaviour. Reliability of this nature cannot be quantified.

To ensure system reliability, we need to keep both views in mind. The complaints about the predominance of mathematical literature in journals and at conferences in the field of system reliability [Oco00] are evidence that research on system reliability puts too much emphasis on the first view. In view of the fact that the most disastrous system failures are caused by unexpected failures, typically stemming from ignorance or neglect of some important observables, we need to put more emphasis on the second view.

EM is a particularly suitable paradigm for investigating the second view of reliability. When building an EM model, our primary concern is not with modelling the abstract observations that characterise the system but the phenomena that sustain the system operation in practice. EM leads to a model that is open-ended and extensible, and can supply a richer model of the environment in which the system is used. In such a model, hidden assumptions and parameters can be explored through intervention of the system developer as a super-agent. In principle, an EM model can simulate the target system in use and generate situations of failure quickly and economically. The reliability of the system can be improved continuously through exploring different ways of interacting with the model.

3.2.3 Unity

Unity is intrinsic to any system because in order to see a thing as a system we need to distinguish it from other parts of the world. Traditionally, this is associated with defining the 'system boundary', which is usually established at an early stage in system development. In that context, the aim is to fix what should be considered to be part of the system and its relationship to non-system entities based on developer's knowledge about requirements of the target system. The term 'system boundary' gives an impression that there is a sharp distinction between system entities and non-system entities. This may lead to a premature focus of attention on building what lies within the boundary whilst ignoring the most significant concern of how the system interacts with the environment.

An EM model for supporting system development does not specify any system boundary. Observables in the EM model are always open for interpretation either as belonging to the system or to the environment. For this reason, the representation of a system in an EM model is very different from that in a traditional system specification. In such a system specification, a system is often represented in terms of parts that presume that appropriate preconditions will be met in the operational environment. With an EM model, the system is represented in terms of observations that presume no precondition on the environment other than – possibly – that observables preserve their integrity. An EM model serves as a description of the system and the environment. Therefore, the unity of a system can always be negotiated by the developer at any time.

3.3 From thinking about systems development activities

System development is a process that involves a variety of human activities. In this section, we shall discuss the EM perspective on system development with reference to three important aspects of development activities: cognitive, collaborative and methodological aspects.

3.3.1 Cognitive aspect

From an EM perspective, one of the most significant elements in system development is knowledge creation. Not all the knowledge created in system development contributes to 'the system'. Significant knowledge may relate to scenarios to be avoided in the system. The knowledge generated in systems development in EM is similar in character to the empirical information gathered by the engineer through experiment prior to systems building. The activity of knowledge creation is associated with the construction of an artefact to embody experiences of the environment from which the system emerges. The process for knowledge creation comprises two important activities:

- **Emergence of knowledge** - This activity begins in the realm of the unknown, as epitomised by primitive interactions with state potentially having a subjective and private significance for the developer. The medium for exploring the unknown is an artefact with which interaction is open and whose interpretation is open-ended. An EM model has the quality of such an artefact. It can embody experience of interaction with a referent that is possibly but not necessarily 'real-world' (cf. Gooding's construal [Goo90]). Through interaction with the EM model, we can identify persistent features and contexts associated with the unknown. Practical skills can be acquired to repeat the observation of these persistent features and contexts. By exercising these practical skills, we can identify dependencies and postulate independent agencies from which generic patterns of interaction and stimulus-response mechanisms are identified as knowledge.

- **Exposure of the unknown** - In this activity, the givens are the knowns. The unknown results from the organization of what is known. Typically the knowledge is associated with familiar experiences, and may be perceived as having a universal relevance, being concerned with objective events and interactions. The knowledge is also associated with patterns of interaction with an EM model which are encountered by the designer in analysis. The aim of the analysis is to find conflicts and expose hidden assumptions within the knowledge. By altering the hidden assumptions, we can obtain new insight about the system. The new insight typically has implications and prompts us to make predictions that expose the unknown. The history of the Periodic Table illustrates the

essential principle. In the Periodic Table, chemical elements are arranged by atomic number and the way in which their electrons are organised. Early versions of the table contained gaps between entries. The existence of elements in the gaps was predicted before they were actually discovered – often at a much later time. The prediction of missing elements is the unknown that requires further investigation.

These two activities are prior to circumscription of any system knowledge. They complement each other as shown in Figure 3.3. The source of the knowledge is the unknown, and the source of the unknown is the knowledge. The generation of the knowledge and the unknown convolve to provide the basis for learning. Therefore, building an EM Model can be seen as an evolutionary process of learning. This is in line with the reflection-in-action paradigm of design discussed in subsection 3.1.2 - in which the design process is seen as a "reflective conversation with the situation" [Sch83]. EM principles and tools provide direct support for this evolutionary design process.

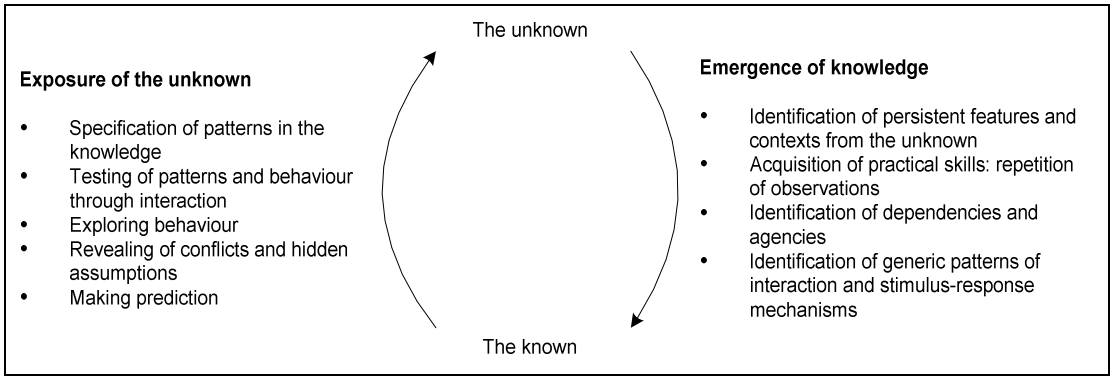


Figure 3.3: An EM perspective on knowledge creation in system development

In EM, we build artefacts to assist our construal of experiences that inform the knowledge of the proposed system. The nature of experiences concerned depends on the current status of the design and purposes of the modeller. Such experiences may relate to the general phenomena that sustain the system behaviour, or the more specific environment from which the system will emerge or in which it will operate. A central activity in development of such construal is the correlation of interaction with the artefact and interaction with the relevant phenomena, environment or prototype system implementation. Such activity is similar in spirit to the 'what-if' interaction with the spreadsheet that is aimed at establishing a close correspondence between interaction with the artefact and interaction with its referent. Its effect is to generate

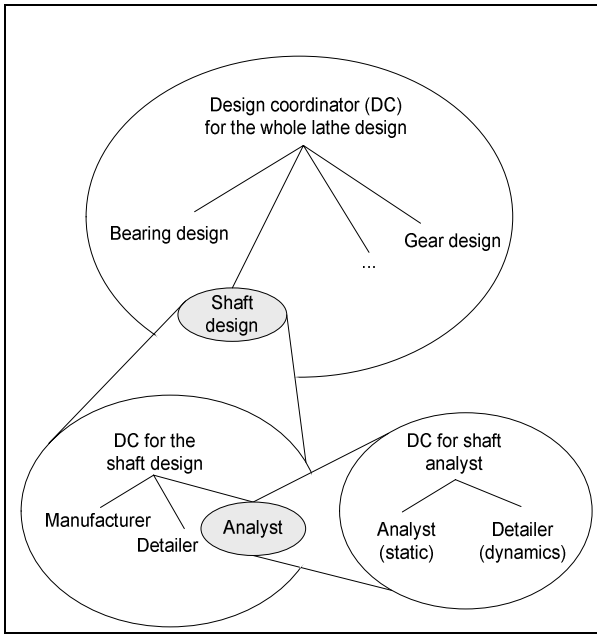
experiences and knowledge representative of all the many viewpoints on the emerging system. As will be discussed and illustrated in more detail in chapter 5, the emergence of a system is associated with a process for bringing coherence to this body of experiences and knowledge. This gives conceptual integrity to the system design.

3.3.2 Collaborative aspect

System development is an activity that typically involves a group of developers rather than one individual. Collaborative system development imposes extra interrelated technological and social concerns. Sharing explanation and understanding of knowledge and experiences is the key to effective collaboration.

In current practice, documentation plays a central role in sharing knowledge. It is where most knowledge of the target system is captured. This is reflected in current groupware, in which documents and document-related processes predominantly define the logical context for collaboration [Mar98a]. However, what can be shared by documentation is only the tip of the iceberg where system knowledge is concerned. Documentation is best suited for recording explicit propositional knowledge which is derived from only a part of stable experience. The source of propositional knowledge, which is practical knowledge and experience embodied in individual developers, remains for the most part unsharable (cf. the Empiricist Perspective on Learning [Bey97]).

In EM, practical knowledge and experience of the target system can be embodied in an EM model. The distinction between conventional documentation and an EM model is not a matter of style but concerns the character of the knowledge being represented. An EM model serves as a medium for sharing practical knowledge and experiences in collaborative system development. In EM, all the activities associated with collaborative system development, including interactions within the system and the meta-interactions between the developers, can be framed as state-change associated with interaction of agents within the DMF. An agent can be a person, a technological entity (e.g. a computer) or a social entity (e.g. a company). This idea was originally developed in the context of concurrent engineering [Adz94b], but we can adapt the framework to any form of system development. The framework includes a hierarchy of agents where agents at upper levels of the hierarchy coordinate agents directly under them. This framework has been studied in [Adz94b] with reference to a case-study: the design of a lathe spindle.



	Agent		
	Analyst	Detailer	Manufacturer
Design Parameter	Typical observation procedure		
min. shaft diameter	measure loads and compute	get nearest standard diameter	measure with micrometer
step sizes, positions	consult bearing, gear sizes	check avail. Bearing, gear sizes	machine tool set up time and cost
static and dynamic characteristics	measure loads, mass, geometry: compute		
material	compute stress conditions	consult catalogue	heat treatment process & measure
spindle operation	compute use, study cutting dynamics		
manufacturing cost		get quotation from contractor	investigate cost of each manuf. Process
material cost		get catalogue cost	actual cost
assemblability	assign tolerances	check standard components fit	measure dimensions

Figure 3.4: Hierarchy of agents in design of a lathe spindle and agents associated with the shaft design (adopted from [Adz94b])

The left-hand-side of Figure 3.4 shows a hierarchy of agents involved in the design of the lathe. The design coordinator is an agent who is responsible for maintaining an EM model that integrates all EM models from the agents directly under him or her. For instance, in shaft design, there are three agents: analyst, detailer and manufacturer. Each agent has their own view of how the shaft should be designed (cf. the right-hand-side of the Figure 3.4). The three EM models of the shaft constructed by the three agents will typically contain complementary and potentially conflicting observations. The design coordinator's job is to merge the observations and maintain a coherent EM model of the shaft.

As the example shows, this framework uses a meta-EM model to describe the hierarchy of agents who are developing an EM model for the target system. The meta-EM model is accessible to all the developers involved – it allows them to maintain a sense of the whole development process and progress – helping them to maintain conceptual integrity of the system architecture. This framework can accommodate the demands of developing not only systems which once deployed usually remain unmodified (such as a dishwasher) but also systems which need constant adaptation (such as information systems). For instance, a change of personnel during system development can be directly reflected in the meta-model.

Two challenges for collaborative system development relate to the decomposition and the synthesis of a system. Where decomposition is concerned, the system is divided into components for each developer. However, after the initial decomposition, the system may sometimes need to be decomposed again in another way due to a change in the requirement. By explicitly representing dependencies and automating dependency maintenance, an EM model admits several different ways of decomposition without any need to compromise the integrity of the system. Where synthesis is concerned, the system components have to be integrated either for final deployment or for testing purpose. System synthesis involves merging concepts from different developers and resolving conflicts between their viewpoints. It involves explanation of one's own work and understanding other people's work (cf. [Gri98]). Developers can get hands-on experience of components by interacting with the EM models of components. Practical knowledge can be transferred in this way.

From the discussions in this and the previous subsections, we can see that developers can simultaneously use EM models both as a medium for reaching consensus and resolving conflicts in system development, and as a playground for individual experiment.

3.3.3 Methodological aspect

One common motivation for studying system development is to identify some universal principles to help improve the effectiveness of system development. Taking software system development as an example, different formal methodologies have been developed to tackle the so-called 'software crisis'. These methodologies usually provide structured and standardized procedures and techniques aiming at improving the software system development process (cf. SSADM [Eva92], the waterfall process model [Beo76] and the spiral process model [Boe81]). However, there is an increasing amount of research literature questioning the usefulness of formal software system development methodologies (e.g. [Bas92, Fit94, Tru00]). Their main criticism is that in practice people do not follow the formal development procedures provided by methodologies (cf. Table 1 in Appendix E). At best, people modify and adapt methodologies to suit their purpose [Rus95, Gre98b]. This argument exposes the same tension between the rational problem solving and reflection-in-action paradigms for describing design activities discussed in subsection 3.1.2. Critics of formal methodologies argue that in practice every process of software system development is unique, and questions the usefulness of standardising any development procedures.

Most experienced researchers and developers are becoming more convinced that system development should not focus on following formal methodologies. For this reason, attention has recently turned away from standardising system development processes to standardising system representations. At present, the Unified Modelling Language (UML) is the most popular system representation standard. UML provides a set of standard diagrams representing different views of a system in development. Two claims have been emphasised in connection with UML: that it is methodology-independent, and that it is suitable not only for the development of software systems but also for that of other non-software systems [Uml02]. It is beyond the scope of this thesis to discuss whether these claims are justified. Nonetheless, the influence of UML to system development in general is huge. Moving from a process-focused approach (where development is driven by applying a formal methodology) to a representation-focused approach (where development is driven by sharing standard diagrams) of system development is a significant shift in perspective in the research and practice of system development.

The EM perspective on system development is different from both process-focused and representation-focused approaches. Formal systems development methodologies are perhaps most useful as guidelines for a new developer with little experience in system development - EM prescribes no general method for system development [Bey98]. A standardised set of representations of a system cannot do justice to the dynamic variety of views of a system from different system developers – from an EM perspective, views of a system should be formed through negotiation amongst developers involved in every aspect of system development. In summary, we cannot in general standardise either the development process or the system representation. In EM, the emphasis in system development is shifted to interaction with EM models. Both process and representation can be modelled in interaction with an EM model. System building should be guided by interaction between actions and situations, and both process and representation should evolve as the development proceeds. Individual and collaborative system development activities are both mediated by interaction with EM models.

In the next section, we shall compare the application of UML and EM in system development with reference to a case study: modelling a dishwasher system.

3.4 Case study: comparing modelling with EM and UML

In this section, we compare and contrast two styles of modelling that might be used in developing a dishwasher system. The aim is to explore the difference between system development based on EM and on modelling using UML. For EM, we use TkEden as our supporting tool; for UML, we use Rhapsody from I-Logix. Both tools provide mechanisms for creating a simulation of the target system interactively. A model of the dishwasher system has been built by using each tool. The processes of building the models have been recorded in subsection 3.4.1. Observations and comparisons are discussed in the subsection 3.4.2. Our simple case study has been chosen with comparison of the two modelling approaches in mind – our main purpose is not to conduct a detailed analysis of both two styles of modelling but to capture major fundamental differences.

A detailed description of UML and Rhapsody is beyond the scope of this thesis. We assume that the reader has a reasonable amount of understanding and hands-on experience of using UML with any one of its supporting tools (e.g. I-Logix Rhapsody [Ilo02], Rational Rose [Rat02], TogetherSoft ControlCenter [Tog02]). We use Rhapsody in this case study mainly because it is more accessible under academic license, and it contains a simple tutorial of how to build a dishwasher model. However, we believe that Rhapsody includes all the common features available in most of the UML tools in the market – these include support for drawing standard UML diagrams, mechanisms for navigating through the model, code generation for the major programming languages, dialogs for filling in properties using forms, etc. On this basis, our findings are also applicable to other UML tools.

The initial requirement of the dishwasher system is adapted from Rhapsody's online tutorial [Ilo02]:

> *The Acme Company requires software for use in the control of a dishwasher. Acme supplies three custom components to control: a tank, jet, and heater. The dishwasher washes dishes by spraying the water stored in the tank using the jet. The jet sprays the water to rinse the dishes and sends pulses of water to wash them. After the dishes are clean, the heater is activated to dry them. One cycle through the dishwasher consists of washing, rinsing, and drying. You can select*

three different cycle modes:

- *Quick – used when guests are on the way and the dirty dishes need to be cleaned quickly.*
- *Normal – used under normal circumstances.*
- *Intense – used when the dishes are extra dirty, and there is more time to let the dishwasher run.*

It should be possible to switch modes at any time. The dishwasher can be started only if the door is closed. If you open the door while the dishwasher is running, operation halts. When you close the door, operation resumes from the point that it was interrupted. To ensure that the dishwasher continues to work properly, it gives an indication for scheduled maintenance after completing a predetermined number of cycles.

The EM model is generated in its entirety starting from this initial requirement. The UML model is built by following Rhapsody's online tutorial. Comparisons are made based on our experience of building these two models.

3.4.1 Two processes for modelling a dishwasher

Development of the EM model

In this section, we shall briefly describe how we built an EM model of the dishwasher system. The definitions shown in Listing 3.1 and 3.2 below are extracted from the model to illustrate our discussion. For a complete listing of all definitions in the model, the reader is directed to Appendix A.

In EM, we can start modelling with whatever observations we deem to be significant in mind. In the case of the dishwasher, we may first observe a door that can be in two states: open and closed (lines 5-8). We know that the dishwasher can be in three modes: quick, normal and intensive (lines 9-13). The rinsing, washing and drying time are dependant upon the mode selected (lines 14-16). For example, at line 14, the rinsing time is 1 second for quick mode, 2 seconds for normal mode and 8 seconds for intensive mode. We can check that the rinsing, washing and drying time is

automatically updated whenever the mode is redefined. Such testing of the model as it is being built is characteristic of EM.

Having introduced the door and the dishwashing modes, we turn to making definitions for the tank, jet and heater. We will only describe the making of the definitions for the tank here (lines 17-29). In addition to being a water container, the tank has two valves: one for filling water and the other for draining water. Lines 18-22 define some states that we can use for defining the water level and valves. Line 23 defines the tank capacity. Lines 24-27 define some interdependent observations about the water level. For example, the water level status can be empty, full or in between empty and full (line 27). Lines 28-29 define the behaviour of the valves. Their definitions show that the washing progress and water level status determine whether the valves are open or closed. We can check that the definitions of the tank are working properly by altering observables they depend on. For example, on redefining water level to 20000 (line25), we should expect the value of `waterLevelStatus` (line 27) to be `tankFull` (line 19).

```
1.  /***********************
2.  AS AN ENGINEER
3.  ************************/
4.  %eden
5.  /* door */
6.  open is 0;
7.  close is 1;
8.  door is close;

9.  /* wash mode */
10. quick is 1;
11. normal is 2;
12. intensive is 3;

13. mode is normal;
14. rinseTime is (mode == quick)?1:((mode
    == normal)?2:8);
15. washTime is (mode == quick)?1:((mode ==
    normal)?2:8);
16. dryTime is (mode == quick)?1:((mode ==
    normal)?2:8);
```

```
17. /* tank */
18. tankEmpty is 0;
19. tankFull is 1;
20. tankSomewater is 2;
21. valveClose is 0;
22. valveOpen is 1;

23. tankCapacity is 20000;
24. waterFlowPerSecond is 50;
25. waterLevel is 1;
26. waterPercent is float(waterLevel) /
    float(tankCapacity) * 100;
27. waterLevelStatus is
    (waterLevel==0)?tankEmpty:((waterLe
    vel>=tankCapacity)?tankFull:tankSom
    ewater);
28. drainValve is (progress==washed &&
    door==close &&
    waterLevelStatus!=tankEmpty)?valveO
    pen:valveClose;
29. fillValve is (progress==go &&
    door==close &&
    waterLevelStatus!=tankFull)?valveOp
    en:valveClose;
```

Listing 3.1: Definitions extracted from the EM dishwasher model

The model is built up incrementally by introducing definitions one by one. In introducing the definitions considered so far, the modeller is acting in the role of an observer who resembles a component engineer. During the modelling process, we notice that there are other roles we can play. We can play the role of a user and define what he or she can observe and act on. Lines 30-49 in Listing 3.2 define three actions for the user – the user can change the washing mode, close the door and open the door. For example, to change the washing mode to intensive in the role of a user we input

"`changeMode('I');`" – this redefines the definition of mode as "`mode is intensive;`". The duration for rinse, wash, and dry will be automatically changed to 8 seconds according to the definitions in lines 14-15.

Another role we can play is that of an interface designer who designs the layout of the dishwasher interface. Lines 50-59 define the appearance and position of the quick-mode button in Donald. For example, line 57 defines that if the machine is in quick mode, the quick-mode button lights up in yellow. In order to test or evaluate the model, we can change the washing mode to quick by inputting "`changeMode('Q');`" to see the quick-mode button light up as a result of automatic dependency maintenance.

```
30.  /**********************
31.           AS A USER
32.  **********************/
33.  %eden
34.  proc changeMode{
35.     para char;
36.       if(char=='Q'){
37.         mode is quick;
38.       } else if(char=='N'){
39.          mode is normal;
40.       } else if(char=='I'){
41.          mode is intensive;
42.       }
43.  }
44.  proc openDoor{
45.       door is open;
46.  }
47.  proc closeDoor{
48.       door is close;
49.  }

50.  /*****************************
51.  AS AN INTERFACE DESIGNER
52.  *****************************/
53.  %donald

54.  #mode buttons
55.  circle quickButton
56.  quickButton = circle({80,290},5)
57.  ?A_quickButton is
     (mode==quick)?"color=yellow,fill=sol
     id":"";

58.  label qLabel
59.  qLabel = label("Q",{80,300})
```

```
60.  /****************************
61.  DISHWASHER COMPONENTS VISUALISATIONS
62.  ****************************/
63.  %donald

64.  # tank visualisation
65.  rectangle theTank
66.  theTank =
     rectangle({100,50},{200,50+waterPer
     cent!})
67.  ?A_theTank =
     "color=blue,fill=solid";

68.  line theTankTop, theTankLeft,
     theTankRight, theFillValve,
     theDrainValve
69.  theTankTop = [{100,180},{200,180}]
70.  theTankLeft = [{100,70},{100,180}]
71.  theTankRight = [{200,50},{200,160}]
72.  ?theFillValveX is
     (fillValve==valveOpen)?180:200;
73.  ?theDrainValveX is
     (drainValve==valveOpen)?80:100;
74.  theFillValve =
     [{theFillValveX!,160},{200,180}]
75.  ?A_theFillValve =
     "color=red,linewidth=2";
76.  theDrainValve =
     [{theDrainValveX!,50},{100,70}]
77.  ?A_theDrainValve =
     "color=red,linewidth=2";

78.  label theTankLabel
79.  theTankLabel = label("water tank",
     {150,200})
```

Listing 3.2: Definitions extracted from the EM dishwasher model

To make it easier to check the states of the tank and its interaction with other components, we can build up a simple Donald visualisation for it. The definitions in lines 60-79 are all we need to visualise the tank. For example, lines 65-67 define the appearance of the water level in the tank as a size-changing blue solid rectangle. The size of the rectangle changes according to the percentage of water in the tank. Figure

3.5 below shows three kinds of visualisation. The top-left segment shows the layout of the dishwasher interface. The top-right segment shows visualisations of the three major components of the dishwasher. The bottom-left is an animated state-chart whose current state is synchronised with the actual state of the dishwasher.

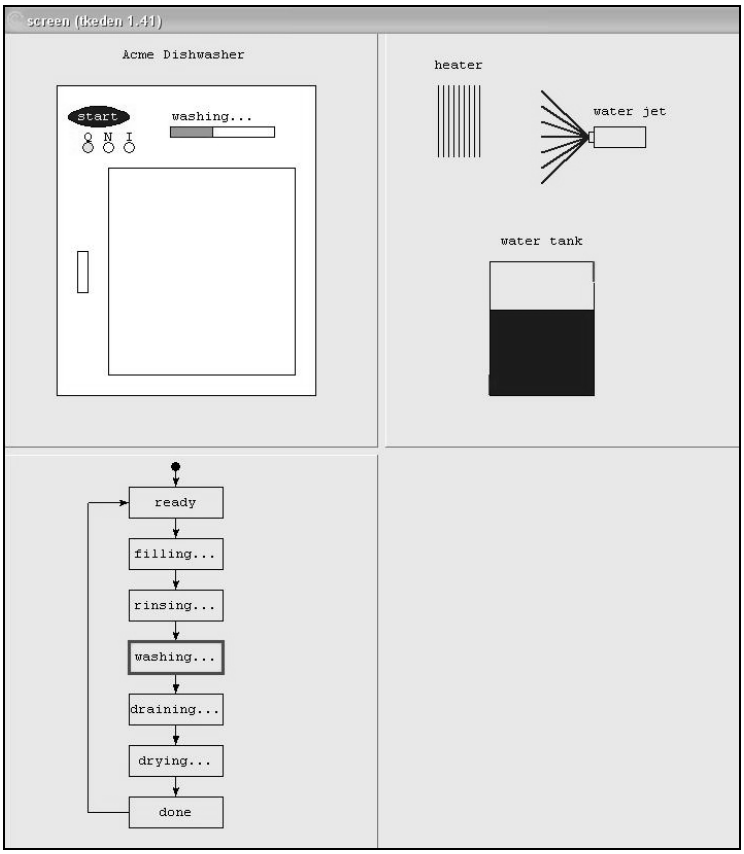


Figure 3.5: Visualisations of the EM Dishwasher model

Appendix C contains an LSD account for the EM Dishwasher model. It illustrates what was in the modeller's mind when constructing this model. It is based on the script model of the dishwasher described above. It can be used as a specification for the actual implementation of the dishwasher.

Development of the UML model

In developing the UML model, we have followed most of the steps in the online tutorial of Rhapsody [Ilo02]. The UML diagrams created are shown in detail in Appendix B. There are four major steps to creating the dishwasher model. Each step

creates UML diagrams of a particular type. Altogether we have created five types of diagram: the Use Case diagram, Class diagrams (Object Model diagrams), Sequence diagrams and State Chart diagrams.

The first step is to create the Use Case diagram. This is based on the dishwasher requirement statement. The diagram shows interactions between the system and external actors. It also defines a system boundary (depicted by the bold rectangle in Figure 3.6). Three use cases are identified. The Wash Dishes and Configure Washing Mode use cases belong to the actor Cleansing. The Service Dishwasher use case belongs to the actor Service. This diagram provides a high level view of how the dishwasher will be used.

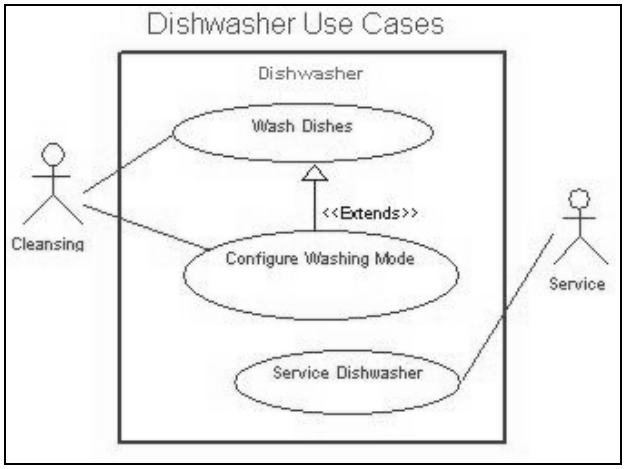


Figure 3.6: The Use Case diagram

The second step is to create Class diagrams. A Class diagram shows the static structure of the system including classes, attributes, operations and relationships like aggregation and inheritance. Figure 3.7 shows a part of a class diagram – we can see that a `Dishwasher` class owns a `Tank` class, `Heater` class and other classes. We can edit all properties of a class by bringing up a class edit dialogue. For example, the display at the bottom-left corner of the figure shows that we are editing the properties of the `Tank` class.

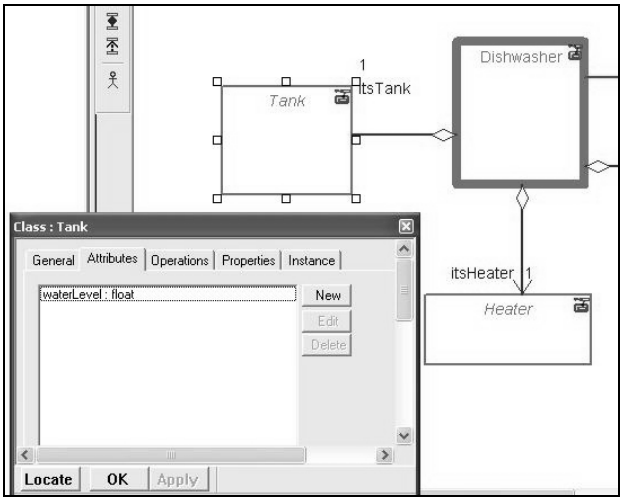


Figure 3.7: Part of a dishwasher Class diagram and a class edit dialogue.

The third step is to create Sequence diagrams. A sequence diagram shows a scenario in the execution of a use case. Typically, it shows a sequence of message passing between different participating objects. Figure 3.8 depicts a part of a sequence associated with the Wash Dishes use case. The arrows in the sequence diagram depict message passing. The time line runs from the top to the bottom of the diagram. In Figure 3.8, we can see two objects, `Dishwasher` and `Tank`, participating in the sequence. First, the sequence is initiated by a event called `evStart()` which is received by `Dishwasher`. The Dishwasher initialises itself by calling `setup()`. Then, the `Dishwasher` sends an event by calling `evTankFill()` to the `Tank` for filling up the Tank. After filling up, the `Tank` confirms it is full by calling `evFull()` back to the `Dishwasher`. All the function calls we specified during drawing the diagram are registered in their corresponding classes. For example, the function `setup()` will be registered in the `Dishwasher` class.

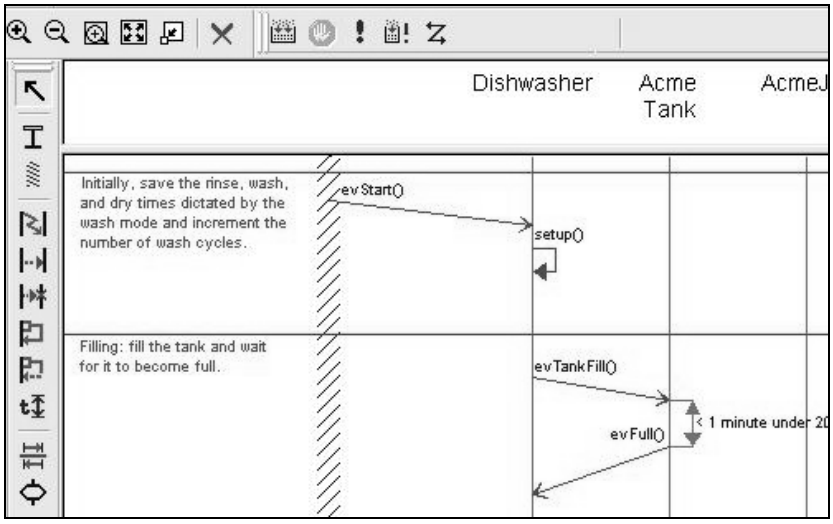


Figure 3.8: A part of a Sequence diagram for Wash Dishes use case

The last step is to create State Chart diagrams for each class in the model. A State Chart diagram defines the behaviour of an object by explicitly specifying all the states the object can be in and the conditions for transition from one state to another. Figure 3.9 shows a State Chart diagram for the `Tank`. There are four states: empty, filling, full and draining. Initially, the `Tank` is in the empty state. When an event `evTankFill` has been generated, the `Tank` goes into a filling state. In the filling state, the `Tank` fills water until the water level equals the tank capacity. The `Tank` then enters the full state. An event `evFull` is generated by the `Tank` to notify the `Dishwasher` that the `Tank` is full. Similarly, the Tank can go on to draining and back to the empty state as a response of some system events.

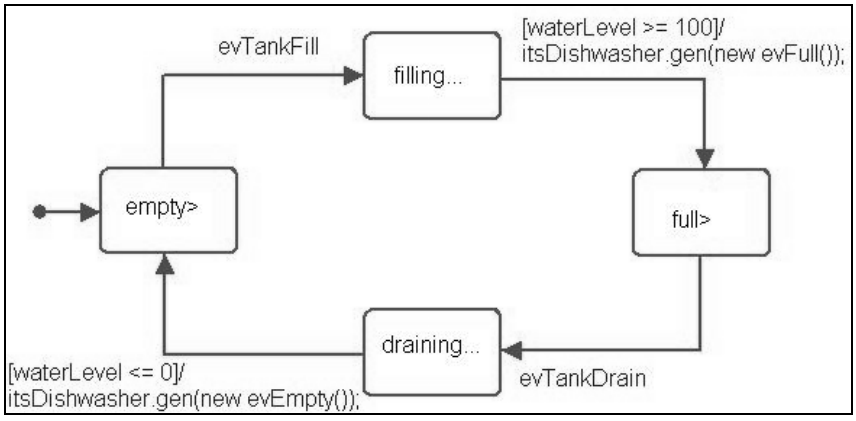


Figure 3.9: A State Chart diagram for the Tank class

After specifying all the UML diagrams, we created the user interface by using Java and linked the interface to the dishwasher system by adding the interface as an `Observer` class. We then compiled the UML diagrams into Java source code and

compiled the Java source code into byte code ready for execution. Figure 3.10 shows a snapshot of the dishwasher system in execution. At the right-hand-side of the figure, we have an animated sequence diagram that shows all message passing between objects as it occurs.

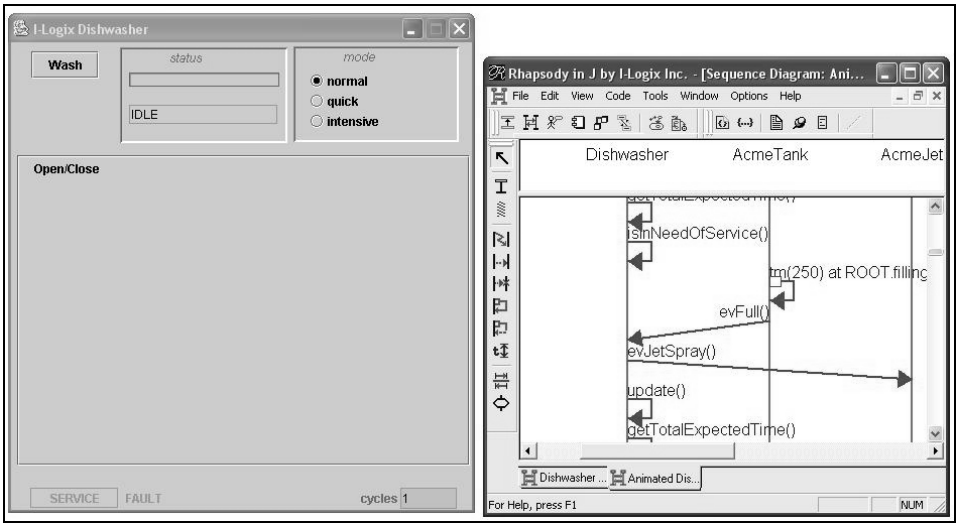


Figure 3.10: A snapshot of running the UML model

3.4.2 Comparisons

In this subsection, we shall make comparisons between EM and object-oriented modelling using UML under five headings: Modelling focus, Interactiveness, Comprehension, Openness and Interfaces.

Modelling focus

We notice that one fundamental difference between the two approaches concerns what is being modelled. EM focuses on modelling the potentially subjective interpretations of the modeller. Observations recorded in an EM model are based upon the modeller's imagined interaction in the roles of agents acting within or on the target system. In EM, the model-building demands more awareness of the situation in which the target system operates. The design of the system emerges from accounting for different aspects of state as viewed by a variety of agents. UML is focused on modelling the structure and behaviour of the system. All the UML diagrams represent views that are most relevant to the system designer.

Subjective use is outside the scope of modelling with UML. Consequently, once the system boundary is established, the system design is typically constructed in a setting that is isolated from its operating environment and use.

Interactiveness

In EM, the representation of specific states is an intrinsic feature of any model. The modeller can always get immediate feedback on enquiries about any particular states for investigation throughout the development of the EM model. The scope for experimental interactions is only constrained by the modeller's construal. Such interactions may involve adding observables without conceptual change of state, or the consideration of states and behaviours outside the scope of normal use. A model offers a real-time response to an experimental interaction. Further experiments can be conducted immediately after changing the model.

UML diagrams are abstract representations of a system that are not primarily intended to be interpreted with reference to a particular system state. The main role of UML is to specify system behaviour. When the modeller makes an experimental change to a UML diagram, it affects the specification of system behaviour. Such experimentation is quite different in character from open-ended interaction with a specific system state. The modeller may be able to change particular system parameters (e.g. the tank capacity or the length of the rinsing phase in the dishwasher) that affect the system behaviour as whole, but experimental interaction with specific system states is constrained by the specified system behaviour (only initial states that lie in the path of a system execution are accessible). Moreover, every time a UML diagram is changed, it has to undergo two phases of compilation before the modeller can get feedback about system states and behaviours. One phase involves translation from diagrams to the source code of a target programming language, and the other involves compiling the source code into an executable program. In addition, code generation from UML diagrams is not fully automatic – the modeller has to manually resolve programming language specific issues (e.g. the declarations of language specific types). As a result, interactive experiments are more difficult to conduct in UML than in EM.

Comprehension

Experimental interaction plays an important role in understanding real world phenomena. The scope and quality of the possible interaction with an EM model facilitates comprehension where specific details of structure, behaviour and performance are concerned. For instance, interaction with the EM dishwasher model allows us to explore the relationship between the behaviour of the fill valve and the filling of the tank. The mechanism can be such that the opening and closing of the fill valve is directly controlled by the dishwashing cycle or by an autonomous stimulus-response mechanism that switches off the valve automatically when the tank is full. Exploration of this nature has no counterpart in the UML model of the dishwasher. Where comprehension is concerned, the function of UML diagrams is to display the predetermined relationships between components rather than to allow the modeller to explore possible alternatives.

A complementary aspect of system comprehension is concerned with understanding the integrity of the system as a whole. In our case study, the modeller cannot grasp this integrity easily when modelling with UML for three reasons:

- The confusing interface – information about a class is scattered throughout different diagrams and interfaces. For example, Figure 3.11 shows two dialogs, one on top of another, that contains information about one class. Bits and pieces of information about the class are scattered throughout tabs and fields whose positions are semantically irrelevant to the model.

- Code generation – In translating UML diagrams to source code, the tool automatically generates additional code, that is not directly relevant to the UML diagrams, making it difficult – if not impossible – for the modeller to understand the source code.

- Consistency between diagrams – There are a lot of dependencies between different UML diagrams. There are few cues to enable the modeller to trace these dependencies easily. This is a well-recognised problem of UML (e.g. [Kim99]). Some suggest introducing automatic consistency checking to the supporting tools [Rob00]. However, even if the consistency can be automatically maintained by a tool, the dependencies cannot be easily comprehended by a human interpreter.

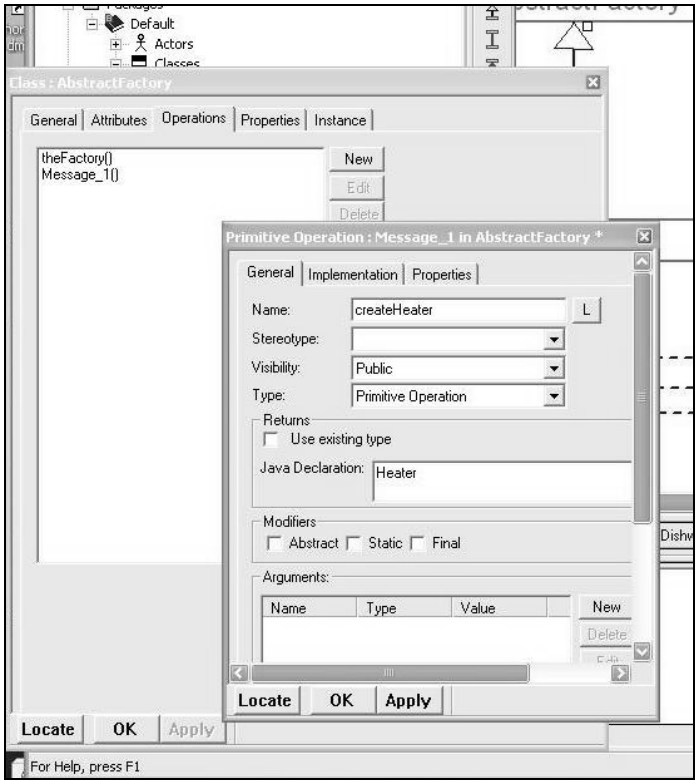


Figure 3.11: Dialogs showing information about a class

Openness

Openness here is related to two principal questions – to what extent does the modelling style support the modeller in:

- exploring hidden problems relating to the requirements as currently framed,
- adapting the model to deal with changing requirements.

In UML, a fixed set for diagrams, representing different viewpoints on the system under development, guides the system modelling [Smo01]. Modelling activity starts with use-case diagrams that define the system boundary. The next step is to draw a class diagram that defines the main components of the system. Classes define what we need to observe by way of attributes of component in order to develop the system.

Other diagrams are constructed based on the use-case diagram and class diagram. These introduce additional properties and constraints governing the interaction

between components until the model is specific enough for implementation. Therefore, modelling with UML resembles a top-down approach to analysis and design. The system is built as a result of *circumscription*. The circumscription takes place from the beginning of the modelling process – use-case diagrams specify the system boundary of interest; class diagrams specify the available observations. These preliminary commitments affect the ability of the model both in accommodating changing requirements and in exploring hidden requirements. The openness of the model is also constrained by the limited viewpoints that a system designer can use.

EM presumes no fixed set of viewpoints. The modelling process involves the identification of observables that may or may not belong to any agents initially. Attributing observables to agents is very different from assigning attributes to classes in UML. The former is a process of discovery and invention whereas the latter is a process of analysis. In EM, agents emerge as groups of observables. The emergence is an ongoing rather than one-off process. It is for this reason that there is no formal syntax for defining an agent for an EM model. In EM, the system is built as a result of *emergence* rather than circumscription. Throughout the process of EM, the system boundary is undefined. Throughout the modelling process, the model always remains open for the exploration of hidden requirements and to accommodate changes in requirements (cf. [Loo98, Loo01, Gog94, Gog96]). Furthermore, EM regards the modeller as a super-agent within the model. The modeller can conduct interaction with any parts of the model to simulate different unexpected exceptional events or conditions in operating the system.

Interfaces

UML does not provide support for specifying interfaces of the target system. Neither can dependencies between states and interfaces of the system be specified by using UML. This means that developers need to use other techniques to design and test the interfaces of the system. In our case study, the interface is created by hard-coding it in Java. EM supports interface design and model-building within the same framework. Dependencies between the states and interfaces are automatically maintained. This feature specifically leads to models for system prototyping that are more stable.

3.5 Aims revisited

As we mentioned in subsection 3.1.3, there are four principal aims in applying EM to system development. In this section, we shall discuss how these aims can be achieved.

- **To promote flexible system design** – An EM model serves as a construal of the system and its situation that is always open to revision. There is no fixed structure within an EM model that is resistant to change - observables in an EM model can be regrouped to reflect structural change without affecting the underlying dependencies between them. As a result, an EM model can typically be easily adapted to accommodate new requirements.

- **To create more reliable systems** – In EM, the target of modelling is not only the system but also the external environments and situations that associated with the system. As a result, system developers can be more aware of the external factors that may affect the operation of the system in actual use. They can simulate various situations of use quickly and economically with the support of the computer-based EM tool. This can help to find identify hidden assumptions that are crucial to the correct operation of the system. The reliability of the system can be tested through interaction with the model. In EM terms, reliability of the system stems from the system developer's experience of stable patterns of observation and interaction.

- **To support the cognitive needs of system development** – An EM model is particularly suitable for recording provisional knowledge and experience that is subjective and unstable in nature. This provisional knowledge and experience is the primary source of creativity and innovation that is important in successful system development. An EM model helps to maintain conceptual integrity in the face of the inherent complexity of the system design. As a result, the system developer can tackle system design problems heuristically.

- **To facilitate collaborative work** – The concepts of EM are domain independent and originate in commonsense. In principle, developers from different disciplines can learn EM easily. An EM model can be used as a medium for supporting communication. Every developer can build EM models of the system within his or her own area of expertise. Different EM models for the same aspect of the system represent different views with potential conflicts clearly exposed. Resolving these conflicts can lead to new system knowledge.

In addition, research into EM provides not only a philosophy for system development but also tools that give practical expression to this philosophy. There are several ways to use EM in system development:

- For the study of existing systems - EM can be used for study existing systems. The product of the study will be an EM model that embodies deep understanding about the system in question. Based on that understanding, we can proceed to improve the existing system or create a new better one. Relevant modelling techniques for this purpose are discussed in [Bey00a].

- For the design of new systems – EM principles and tools can be use to design new systems. The EM model created by the process of design can be circumscribed by producing a LSD specification. The LSD specification can be used as a blueprint for implementing the system.

- For the implementation of new software systems – If a software system is to be developed, we can create an EM model either to serve directly as the new system or as a prototype system from which a suitable conventional system can be generated semi-automatically by translation. In either case the EM model that underlies the software system is very flexible and adaptable to change.

3.6 Summary

In this chapter, we have discussed the EM perspective on system development from two different directions. The first direction involves thinking about the relationship between EM for system development and the intrinsic properties of systems. The second direction involves thinking about EM in relation to important aspects of system development activity. We have also described a case study of building a dishwasher model by using both EM and UML. As a result, we have compared the differences between EM and UML styles of modelling systems. We have discussed the aims of applying EM to system development and how we can achieve these aims.