5 Before Systems: Conceptual Integrity

In his famous book, The Mythical Man-Month [Bro95], Brooks contends that "conceptual integrity is the most important consideration in system design". This chapter endorses and elaborates this idea, and contends that obtaining conceptual integrity is essential before a coherent system conception can be established. We believe that, in successful system design, conceptual integrity emerges from activities that are prior to system identification. Along with the discussion we shall explain how the EM perspective on system development helps to address the issues of obtaining conceptual integrity.

This chapter is organised as follows. Section 5.1 explores the meaning of conceptual integrity. Section 5.2 discusses the importance of conceptual integrity in system development, and identifies issues that arise in maintaining the conceptual integrity of a system design. Section 5.3 describes how EM can help system developers to maintain conceptual integrity by addressing the issues identified in section 5.2. Section 5.4 illustrates the discussion with reference to a TkEden model of a railway. Section 5.5 compares EM with other technologies that aim to help system developers to maintain the conceptual integrity of a system design.

5.1 What is conceptual integrity?

In the context of system development, the term 'conceptual integrity' was first introduced by Brooks in the 1975 edition of his book The Mythical Man-Month [Bro95]. Drawing on much experience of system development, Brooks contends that:

> *"Conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas…Conceptual integrity in turn dictates that the design must proceed from one mind, or from a very small number of agreeing*

resonant minds." [Bro95]

Our discussion in this chapter aims to endorse and elaborate the importance of conceptual integrity in the design of systems, and offers EM to help developers to maintain conceptual integrity in a system design.

Brooks associates 'conceptual integrity' with the 'unity of design' but offers no further satisfactory explanation on what conceptual integrity is. To appreciate the importance of conceptual integrity, we need to take a closer look at its meaning. The word 'conceptual' is associated with the cognitive process of concept forming that involves the conscious recognition and identification of elements of our experience; and the word 'integrity' is associated with the idea of 'being integrated' or 'being one'. The idea of a coherent whole is reflected in the way that 'having integrity' is used to describe something that always conforms to one's expectations – there is an implicit reference to future events. We describe something as having conceptual integrity, if the concepts formed from our *experience* of the thing can reliably determine the future events which are associated with the thing (there is no surprise). In *obtaining* conceptual integrity, we are concerned with the emergence of concepts (representing the rational world) from experience (representing the empirical world). As the American philosopher, William James, points out in his Essays on Radical Empiricism, first published in 1902:

> *"Experiences come on an enormous scale, and if we take them all together, they come in a chaos of incommensurable relations that we cannot straighten out. We have to abstract different groups of them and handle these separately if we are to talk of them at all." [Jam96]*

Raw experiences are potentially confusing or incoherent as "they come in a chaos of incommensurable relations". To make sense of them, we need to abstract groups of relations between them and study them separately. Abstracting groups of relations from experiences is the embryonic concept-forming activity. The concepts formed may conflict each other. Obtaining conceptual integrity is a process of removing conflict. To remove conflict, we need to access the raw experiences that originally informed our concepts – to explore them, to understand, to compare and to analyse. This idea is depicted in Figure 5.1. At the left-hand-side of the figure, two groups of concepts are formed by abstracting relations from raw experiences. To obtain conceptual integrity, we need to combine the concepts into one group of unified concepts. This involves the resolution of potential conflict between concepts so that

they acquire coherence.

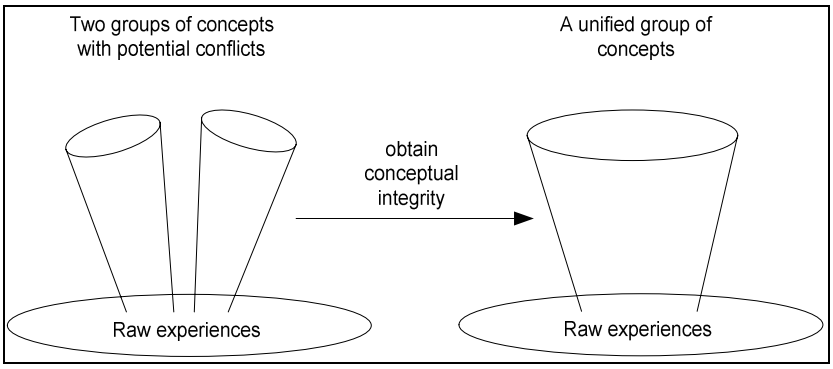

Figure 5.1: Obtaining conceptual integrity from raw experiences

Significant features of the above observations that relate to the nature of conceptual integrity include:

- Conceptual integrity is something appreciated by an observer. This is because we cannot talk about experience without reference to the observer.
- To obtain conceptual integrity, we need to resolve conflicts between concepts. This involves access to the raw experiences that inform these concepts.
- A central issue in obtaining conceptual integrity is the potential incoherence of raw experiences.

The process of obtaining conceptual integrity is closely associated with heuristic cognitive activities that are hard to formalize (cf. Naur's characterisation of a science, which regards "coherent description as the core of the scientific/scholarly acitivity" [Nau01]). Since, as mentioned in the last chapter, EM has the potential to support heuristic problem solving, we believe that EM can help in obtaining conceptual integrity. An illustration of this process can be found in section 5.4

5.2 Conceptual integrity of system design

System development is located at the intersection of formal and informal, objective and subjective, and technical and non-technical activities (cf. [Sun99a, Wes97]). It is difficult for developers to maintain the conceptual integrity of the system design. By 'system design', we mean the current state of the design of the system under development. A system design evolves throughout the system development process. Changes to the system design are usually made concurrently by different members in the system development team. This makes maintaining conceptual integrity of the

system design even more difficult. But conceptual integrity is surely necessary because it is essential for:

- innovation – with deep understanding of the current state of the system design, the developers can relate concepts in the system design more easily to new ideas;
- flexible system design – a system design with conceptual integrity can be more easily adapted to possible change of requirements;
- effective project management – better knowledge about the status of the design makes it easier to adjust available resources.

Conceptual integrity is of the essence in system development in that what lacks conceptual integrity cannot be easily perceived as a coherent system. The difficulty of maintaining conceptual integrity can be understood in relation to a tension involved in system development depicted in Figure 5.2 below. This tension exists between distinguishing and synthesising different viewpoints at both personal and interpersonal levels.

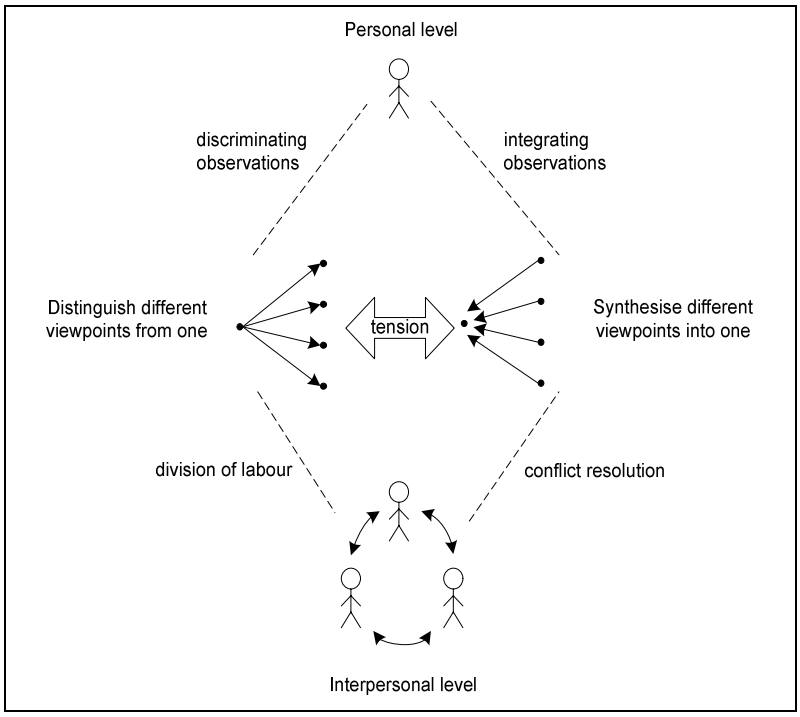

Figure 5.2: A major tension in system development

At a personal level, distinguishing viewpoints involves discriminating different observations from apparently the same experience; synthesising viewpoints involves integrating different observations as a unified experience. At the interpersonal level, distinguishing viewpoints is the key to the division of labour; and synthesising

viewpoints involves conflict resolution. The tension between distinguishing and synthesising viewpoints always exists in system development – we want to distinguish viewpoints on an experience but at the same time see them as a unified whole. The potentially incoherent experience that originates from trying to reconcile these viewpoints is the main obstacle to obtaining conceptual integrity. The key questions that need to be answered in relation to obtaining conceptual integrity in system design are:

At a personal level,

- How can we represent experience that is potentially incoherent?
- How can we effectively abstract concepts from experience?
- How can we resolve inconsistency in concepts?

At the interpersonal level,

- How can we 'share' experience and concepts?
- How can we resolve conflicts and reach consensus?
- How can we represent alternative views?

These are difficult questions to answer because, as mentioned above, the process of obtaining conceptual integrity is hard to formalise. For this reason, conventional formal methodologies have limited applicability. To address the questions, we need to have a human-centred approach to system development that emphasises the creative nature of the cognitive process. We believe that the principles and techniques of EM offer a plausible solution that can satisfactorily address these questions.

5.3 EM for maintaining conceptual integrity

The central idea applying EM to help developers to maintain conceptual integrity in system development is to construct Interactive Situation Models (ISMs). ISMs can be used to represent situated experience acquired through a variety of system development activities. An ISM serves a role in knowledge representation in the sense that 'one experience knows another' [Bey99]. In introducing the concept of an ISM, Beynon and Sun [Bey99] cite writings on philosophy [Jam96], linguistics [Tur96] and experimental science [Goo00] that reflect the importance of negotiating meaning through interaction with artifacts where processes of explanation and knowledge creation are concerned. They point out that "[w]here formalisms aim at freedom from ambiguity and independence of agency and context, experiential representations rely

essentially upon the engagement of the human interpreter". This highlights the limited role that formalisms can play in system development. The EM approach to overcoming this limitation is to use an ISM to embody experience (potentially incoherent) that human interpreters can abstract and share by interacting with it, and that may eventually lead to conceptual integrity in a system design.

In EM, there is a direct correspondence between observables, dependencies and agents in the external world and variables, definitions and actions in an ISM. It is this faithful metaphorical representation of real world entities that makes an ISM a powerful representation of what we observe and experience. Specifying variables, definitions and actions in an ISM is *not* primarily a rational process – because the modeller does not have to give a logical justification for their existence in the model – rather it is an empirical process in which observations are faithfully recorded. On this basis, even incoherent experience can be embodied in an ISM. By analogy, an ISM resembles a draft sketch for a painting (cf. [Ras01]). Its purpose is primarily concerned with *forming concepts* rather than *specifying concepts* –not all the lines in the sketch will eventually remain in the final painting; nevertheless, every line in the initial sketch contributes – even if only indirectly – to the final painting at some stage in the painting process. EM is concerned with identifying and exploring relevant observations prior to formulating concepts from which the system emerges. By its nature, this process of identification demands that we explore things that are eventually deemed to be outside the final system.

As mentioned earlier, James's prescription for making sense of experience is "to abstract different groups of [relations] and handle these separately". In EM, activity of this nature can be performed by extracting groups of definitions from an ISM and studying them separately. An effective way to study concepts is to experiment with them in the manner that is similar to the use of a spreadsheet. A typical use of a spreadsheet involves correlating the real world situation with the state captured in the spreadsheet cells. For instance, a spreadsheet can represent the financial situation of a company. A change in the value of a cell may reflect a change in the real world situation of the company. An accountant can perform many 'what-if' type of experiments to explore a variety of situations in which the company will be affected. By interacting with a spreadsheet, one can obtain experience that reflects the experience of changing the real world financial situation of the company. Based on this experience, one can analyse existing concepts and form new concepts. Using an ISM resembles using a spreadsheet in this respect, but provides more general functionalities that are suitable for studying concepts in potentially any domain. For

this reason, an ISM is a test-bed within which the concepts of a system design can be studied, which is essential for obtaining conceptual integrity of the system design.

Since an ISM embodies experience and concepts, a developer can share experience and concepts with others by sharing an ISM. Unlike a static document, an ISM can be shared and studied by others interactively. For this reason, an ISM is like a concrete physical prototype of a product which one can study by directly experiencing it rather than merely in an abstract fashion.

Where collaborative work is concerned, the collaborative definitive modelling in the DMF provides an agent hierarchy structure within which ISMs can be shared and conflicts can be resolved. The detail of the structure and mechanisms involved has been already discussed in chapter 2, and chapter 3 illustrates the idea with reference to the design of a lathe spindle.

To show that the EM perspective on system development is highly relevant to the conceptual integrity of a system design, we can reinterpret the concerns of system development discussed in chapter 3 in terms of conceptual integrity.

- Complexity – In system development, a system design will be expressed with reference to many different ideas and viewpoints. The viewpoints are not necessarily or typically consistent when taken in conjunction. In other words, they all come together with such a degree of complexity that is difficult to make sense of them. If we are to achieve conceptual integrity, we need to be able to represent complex experiences of this sort that lack conceptual integrity. At present, we ‘represent’ such experiences in a rather incoherent way using documentation in both natural and formal languages, possibly supported by prototypes, and manage sense making through documented meetings between developers. If we cannot represent such experiences effectively, we cannot begin to resolve the conflicts that inhibit sense making and are tempted to resort to simplifying compromises. By using EM, we aim to achieve conceptual integrity of a system design *without compromising complexity*. There is evidence that achieving conceptual integrity without compromising complexity is possible. For instance, Brooks [Bro95] cites the cathedral at Rheims as an example of a complex structure having conceptual integrity.

- Predictability – The perception of conceptual integrity is associated with confirmation of expectation. Things have conceptual integrity if they do not

surprise us, in the sense that what we know about the system is a reasonable guide to what we do not know. It should be noted that this predictability, that is fundamental to conceptual integrity, manifests itself in observation and interaction. The fact that all concepts are expressed by using a set of common modelling abstractions (as e.g. in the proposals for 'seamless' software system development discussed in [Lid94, Pai01]) does not necessarily mean that a system will exhibit conceptual integrity. For example, representing a set of programs using a functional programming paradigm arguably does not guarantee their conceptual integrity. Functional programming is well-suited to the representation of simple programs whose entire significance is captured in their input-output relation. It cannot easily meet the need to embrace other viewpoints on programs, such as arise where complex interaction or visualisation is involved. For this reason, where programming is concerned, the use of the functional paradigm entails trading complexity for conceptual integrity.

- Unity – The coherence of things that have conceptual integrity means that we can experience them as 'being one thing'. Grouping observables into objects as in OOA is one way to try to bring unity to a complex phenomenon, but the coherence and integrity of an object is achieved by extracting it and viewing in isolation from a single viewpoint. On the other hand, the coherence needed for conceptual integrity in a system design needs to embrace many viewpoints. This is similar to the situation in everyday experience, where the integrity of observables is not simply confined to object associations but extends to more subtle dependencies. As explained and illustrated with reference to modelling a door in subsection 4.2.2 of Rungrattanaubol's thesis [Run02], "in modelling with definitive scripts, dependency is the feature that gives integrity to [the] diverse representations of a 'single' object". The key feature of modelling with definitive scripts is that it allows us to add observables within conceptually the same state. In other words, adding a definition to the script need not mean changing the state to which it refers, but rather enriching our perception of the state to which it refers. With such use of definitions, dependencies between observables are seen to be the mediators of unity, expressing 'what belongs to what'.

- Cognitive aspect – In the convolving generation of knowledge, systematic elements emerge. These elements sometimes make sense individually but are incoherent when combined. Potentially, subject to suitable compromises and shifts in perspective, all the requisite systematic elements can be made coherent (cf. Naur's idea of science as 'coherent description of aspects of the world'

[Nau01]). Conceptual integrity emerges in system development through bringing coherence to groups of systematic elements taken together – as an incremental and evolving process.

- Collaborative aspect – Even when each individual developer's view has integrity, we still need to bring them together to realize a coherent overall design in the spirit of Brooks's 'one mind'. The DMF provides collaborative definitive modelling in which a hierarchy of agents/developers interact with each other by sharing ISMs to achieve a coherent and consistent representation.

- Methodological aspect – Conceptual integrity cannot be imposed by formal methods or business processes. Obtaining conceptual integrity is a heuristic cognitive process. EM provides an observation-led approach that is not associated with preconceived ideas about how the observables in the domain should be organized. For instance, whereas OOA obliges the modeller to identify and maintain object boundaries throughout the system development process, EM does not favour any specific whole-part decomposition of a system under development unless or until one emerges. The motivation for such decomposition may come from the properties of the domain ('thinking about the system') or from the demands of distributed development ('thinking about system development activities'). EM does not separate the phases of the development activities according to a preconceived pattern. This promotes conceptual integrity through working practices that are in sharp contrast to a classical development process, where the contributions made by separate design participants at the various stages are largely independent.

We summarise this section by offering answers to the questions asked at the end of section 5.2:

At a personal level,

- How can we represent experience that is potentially incoherent? – by using ISMs raw experience can be faithfully represented. EM is not primarily concerned with modelling a system but facilitating construals of situations in the environment from which the system emerges.
- How can we effectively abstract concepts from experience? – in EM, observation, interaction and experiment are the key to the development of concepts.
- How can we resolve inconsistency in concepts? – by extensive interaction and

experiment with the model, inconsistency can be resolved.

At the interpersonal level,

- How can we 'share' experience and concepts? – an ISM resembles a concrete physical prototype in that it can be shared amongst developers, but offers a far richer level of conceptual support through interaction than a conventional prototype. By interacting with a shared ISM, developers can share experience and concept.
- How can we represent alternative views? – the modeller is able to add observables to an ISM without changing its conceptual state. Observables or groups of observables representing alternative views can be linked together with underlying dependencies.
- How can we resolve conflicts and reach consensus? – the hierarchical collaborative definitive modelling structure in the DMF helps developers to resolve conflict and reach consensus in a manageable way.

In short, EM aims at enhancing the scope for interaction in system development by allowing experience and concepts to be embodied in computer-based models that can be shared, explored and discussed dynamically to facilitate the maintenance of conceptual integrity in a system design.

5.4 Case study: the Railway model

In this section, we shall describe the Railway model and use it to illustrate the ideas discussed in earlier sections. The model was developed by Y. P. Yung in the EM research group, and has been used to illustrate various aspects of EM research in other contexts [Bey90b, Adz94a, Adz94c]. It is a TkEden model with scripts of a variety of definitive notations. The model contains a number of submodels which will be described one by one as the discussion develops.

The Railway model contains a model for designing track layout. Figure 5.3 shows the visualisation of the track layout model. The track segments are arranged to form two circuits one inside the other. The two circuits are connected by the pair of sets of points located at the top of the figure. Although the track layout is simple, the definitions for track segments are based on standard track segments from a track catalogue - so that the length and curvature of each piece is based on a standard specification. Listing 5.1 shows an extract from Donald definitions that defines one of the segments in the track layout. The openshape ST226 defines a standard

prototypical segment with the product code ST226. The openshape A9 defines a segment on the track that is based on a transformation of the segment ST226.

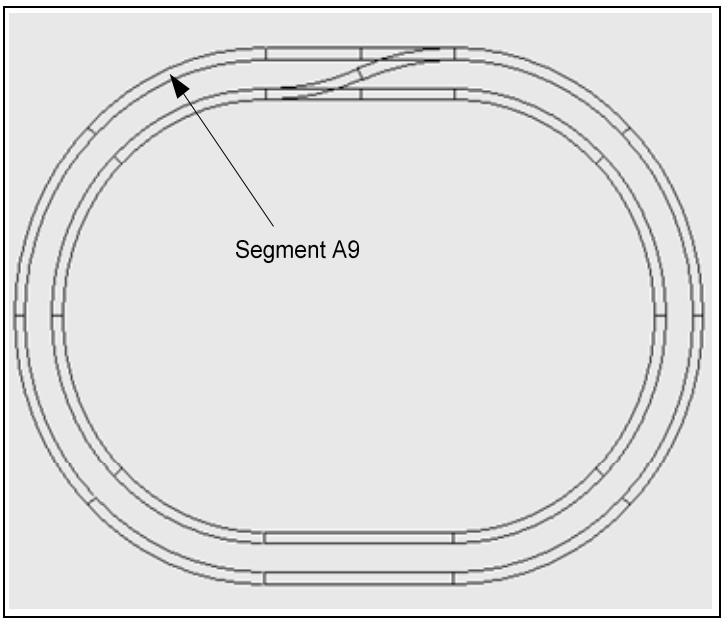

Figure 5.3: The track layout model

```
# standard prototypical curved track segment
openshape ST226
within ST226 {
     real angle
     angle = pi div 4
     real x1, y1, x2, y2
     x1 = (~/Radius2 - ~/TrackWidth div 2) * sin(angle)
     y1 = x1 * tan(angle div 2)
     x2 = (~/Radius2 + ~/TrackWidth div 2) * sin(angle)
     y2 = x2 * tan(angle div 2)
     line f, e
     f = [{0, ~/TrackWidth div 2}, {0, -~/TrackWidth div 2}]
     e = [{x1, y1 + ~/TrackWidth div 2}, {x2, y2 -~/TrackWidth div 2}]
     arc l, r
     l = [{0, ~/TrackWidth div 2}, {x1, y1 + ~/TrackWidth div 2}, \
          -angle * 180 div pi]
     r = [{0, -~/TrackWidth div 2}, {x2, y2 -~/TrackWidth div 2}, \
          -angle * 180 div pi]
}
# A9 track segment based on transformation of ST226
openshape A9
within A9 {
     point start, end
     real inDir, outDir
     start = ~/A10/end
     inDir = ~/A10/outDir
     shape track
     track = trans(rot(~/ST226, {0,0}, inDir), start.1, start.2)
     end = rot(start, start + {~/Radius2 @ inDir + pi div 2},~/ST226/angle)
     outDir = inDir + ~/ST226/angle
}
```

Listing 5.1: Definitions of a track segment based on transformation from a standard prototypical part. An extract from the script producing Figure 5.3.

The use of standard track pieces in Figure 5.3 resembles the use of physical

prototypes. In this case, the designer chooses to use small-scale replicas of real standard track segments. The task involves selecting the right pieces and connecting them together in order to experiment with possible layout options. By its nature, the process is interactive and informal. The merit of a physical model over a mathematical model is that the designer can get concrete experience from the model by interacting with it - this helps the cognitive process of understanding and stimulates creativity (cf. Gooding's construals [Goo90]). The EM model of the track layout has the same quality as a physical model. The designer can interact with the model by making definitions. The Donald visualisation of the model gives immediate feedback to the designer's actions. Of course, the physical model has many attributes and behaviours that are not represented in the EM model, such as the weight of each segment and the rate of track expansion on heat under sunlight. Identification of additional attributes and behaviours is part of the design process. In EM, the designer can always extend the model by adding extra definitions to reflect his understanding and to embody practical experience obtained through interacting with the model. In this respect, an EM model has an significant advantage over a physical model. For example, if there is a budget for the cost of the track, the designer can introduce a price for each standard segment and add definitions for maintaining a total cost for the design. In the case of a physical model, it is necessary to manually maintain a separate cost model in parallel. The synchronisation between the two models has to be performed manually.

In a real situation, the designer typically has to consider integrating a wide variety of such different viewpoints. It is difficult for the designer to maintain the conceptual integrity of the whole design. In EM, such models can be integrated into one model in such a way that the selection of segments in the layout is automatically reflected in the total cost.

The model depicted in Figure 5.3 and the associated Donald definitions only specify the geometric appearance of the track. To further appreciate the potential power of integration, we can consider the segment connectivity model depicted in Figure 5.4. This model is a part of the Railway model that describes the combinatorial relationships between track segments. It defines a topological layout of the track. Such a layout is essential because the connectivity of track affects the possible paths that the trains can travel. We need a representation that expresses the relationship between adjacent track pieces more explicitly than physical coincidence of endpoints on the display. The topological layout depicted in the left-hand-side of the Figure 5.4 serves this purpose.

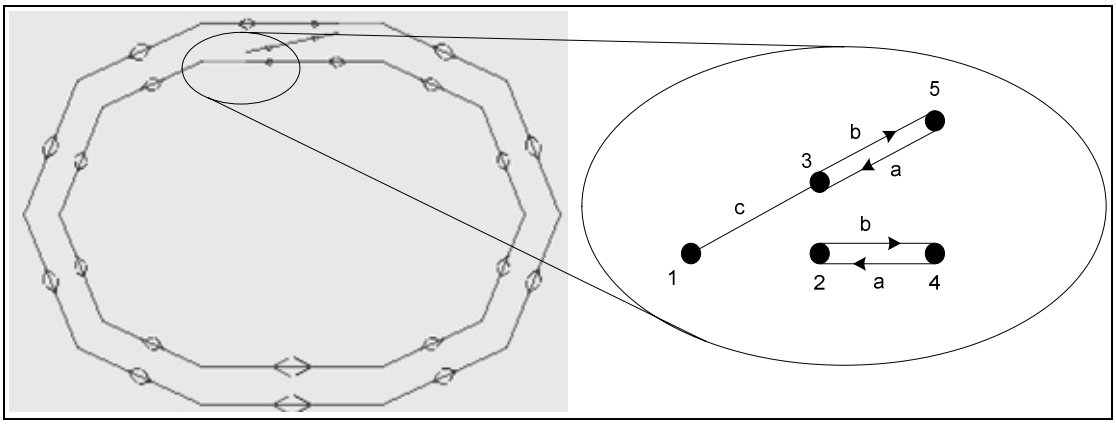

Figure 5.4: The connectivity model

```
1.   mode Point1 = 'abc'-diag 5   #points have 5 vertices and 3 colours

2.   Point1Stts = 1   #status of the points, 1 = closing the loop
3.   #defining the edges

4.   a_Point1{1}=1; b_Point{1}=1; c_Point1{1}=if(Point1Stts==1) 2 else 3
5.   a_Point1{2}=2; b_Point{2}=4; c_Point1{2}=if(Point1Stts==1) 1 else 2
6.   a_Point1{3}=3; b_Point{3}=5; c_Point1{3}=if(Point1Stts==1) 3 else 1
7.   a_Point1{4}=2; b_Point{4}=4; c_Point1{4}=4
8.   a_Point1{5}=3; b_Point{5}=5; c_Point1{5}=5

9.   Scale = 100                 #scaling factor for locating vertices
10.  Point1!1=A!1                #defining locations of the vertices of Point1 in
11.  Point1!2=A!1-[Scale/2,0,0]  #terms of the location of the first vertex of
12.  Point1!3=A!1-[Scale/2,Scale/10,0] #A, the graph representing the outer loop
13.  Point1!4=A!1-[Scale,0,0]
14.  Point1!5=A!1-[Scale,Scale/5,0]
```

Listing 5.1: Definitions that specifies the digraph at the right of Figure 5.4

The topological layout uses a directed graph with coloured edges to describe the connectivity between segments. Each vertex of the graph represents a track piece and each edge represents a join of connected track pieces. In the graph, the direction of an edge represents a direction for the traversal of the associated track segments. The right-hand-side of Figure 5.4 depicts one of the sets of points that link the two circuits - it contains 5 vertices (numbered from 1 to 5) and edges of 3 colours (colour a, b and c). The corresponding definitions are shown in Listing 5.1. The a-coloured edges represent the path that a train can travel in an anti-clockwise direction in the circular track. Similarly, the b-coloured edges represent the path in a clockwise direction. The c-coloured edge is an undirected edge that conditionally links to one of two vertices depending on the current status of the points (i.e. vertex 1 connects to either vertex 2 or vertex 3). The definitions in lines 2-6 of Listing 5.1 serve this purpose.

The track layout model and the segment connectivity model complement each

other as two alternative views for the track design. Together they also provide the necessary context for adding the third submodel: a train simulation model. One feature for this model is that it introduces two trains as autonomous agents. Figure 5.5 depicts an integrated view of three models. In the figure, the positions of two trains are represented as bold lines on the track layout model. The current direction of a train can be inferred from the circle at the tail of the train. The bottom of the figure shows the interfaces for the signalman who controls the points and the two drivers of the trains. As the interface shows, each train can travel in a clockwise or an anti-clockwise direction, and at any time the train can be stopped by pressing the stop button. Moreover, the speed of train movement in the simulation is governed by a clock agent.

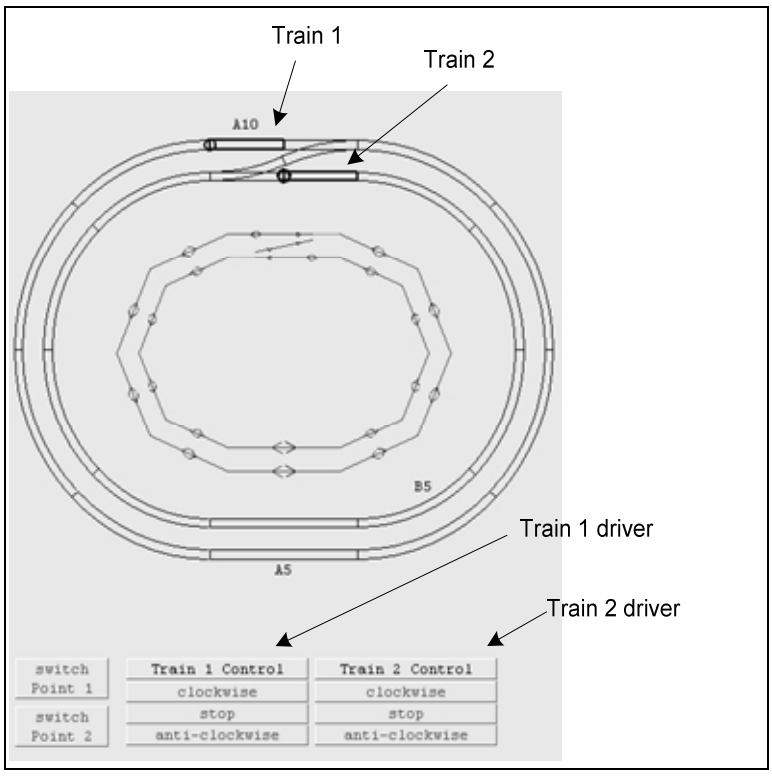

Figure 5.5: An integrated view of the track layout model, the connectivity model and the train simulation model

Adding a train simulation model to a track design potentially enables us to study it in a very rich context. For example, we can explore the consequences of track failure. Where normal operation is concerned, we can also use the integrated model to study the co-operation between the drivers and the signalman.

The last submodel integrated into the Railway model is the train station model. This model simulates the train arrival and departure protocols with reference to one of

the trains (namely Train 1 in Figure 5.5) and the three stations (namely A10, B5 and A5 in Figure 5.5.). Figure 5.6 shows the entire visualisation of the Railway model. The right-hand-side of the figure depicts two stations: the most recent departure station for Train 1 (namely A10 as displayed at the top of Figure 5.6) and its next scheduled arrival station (namely B5 as displayed below). Passengers on Train 1 and at the stations A10 and B5 are also depicted. Each station has a stationmaster who communicates with a guard on the train to manage the train arrival and departure. The protocols that govern the interaction of the various agents within the train station model were derived from an LSD account listed in Appendix J.

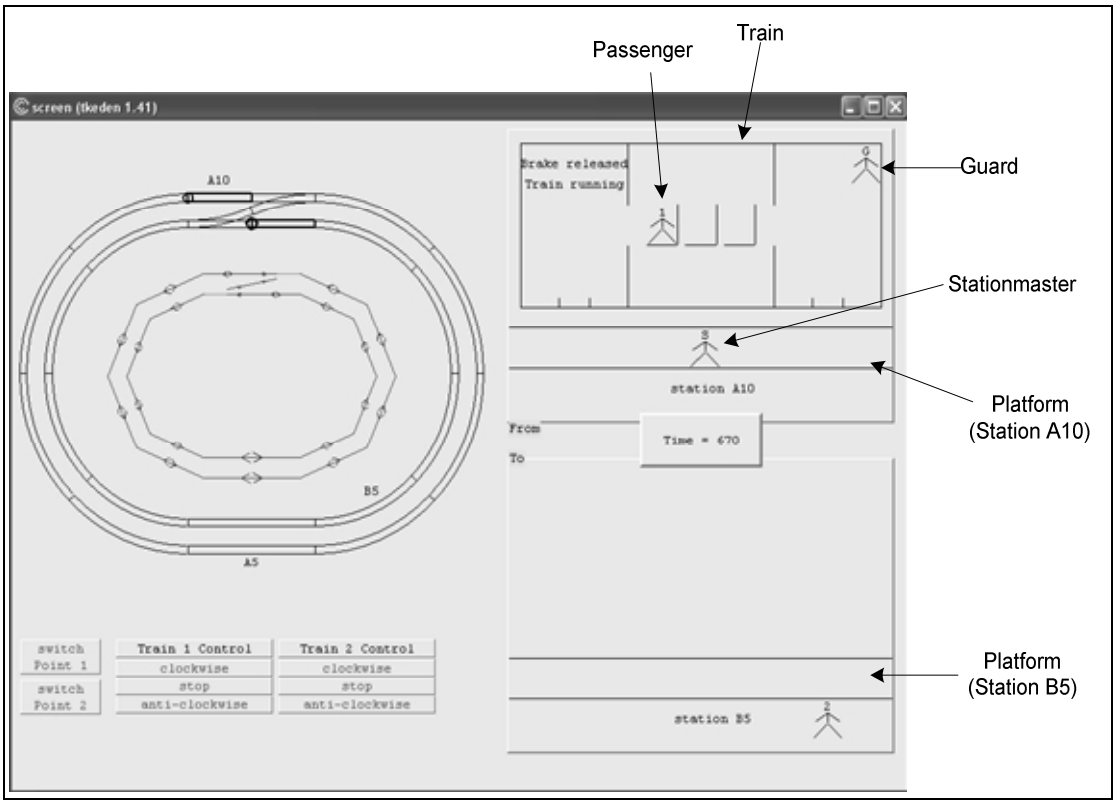

Figure 5.6: Visualisation of the entire Railway model

The Railway model itself has no clear purpose. It is an EM model that illustrates the possibility for embodying incoherent experience that lacks conceptual integrity. Bits of the model make sense independently but not when taken as a whole – the model can partially sustain a whole variety of interpretations, but every interpretation is problematic in some respect if the entire model is taken into account. For example, one obvious interpretation for figure 5.5 is that it is a model railway operated via the interface supplied by Train 1 and Train 2 controls. On the other hand, the visualisation of the train itself resembles the visualisations that are used in practice in electronic signal boxes to indicate which real life 'track segments' are currently occupied. In this

case, there is a conflict between whether we interpret a track segment in Figure 5.5 as a standard piece of model railway track or as representing a segment of a real track possibly several miles long that can (according to normal real-life conventions for railway operation) be occupied by at most one train at any time. To resolve this conflict, we should need *either* to develop a track layout that more closely resembles a context for real world railway operation (e.g. match the layout to part of a particular real world railway network) *or* to visualise the trains as spanning several track pieces, as would be realistic in the model railway scenario.

Another example of a conflict relates to agency. In the model railway model in Figure 5.5, the trains themselves do not 'really' have drivers. It is the modeller or user who drives the train through operating the simplified control interface. Whether this is appropriate depends on the purpose for using this model. A track layout designer will probably find that a simple control interface is all he needs for testing different design layouts. However, a railway safety supervisor might find that it is crucial to model the drivers of the train in more detail. For instance, a driver's reaction to signals is very significant where the safety of the railway is concerned (see e.g. the EM railway accident simulation model described in [Sun99b] which involves extensive exploration of a historical railway accident involving three drivers and two signalmen).

In developing a complex railway system, division of labour is unavoidable. There are many people involved in different aspects of the system. This diversity of concerns leads to conflicts in the design. A railway model such as this one facilitates the sharing of understanding across different members involved in the development. In this respect, the railway model acts as a medium for communication between developers.

When the train arrival and departure model was originally developed, there was no visualisation for the actual stations and the physical track between them. In an early prototype of the integrated model, passengers were observed to get off the train in unexpected contexts. This illustrates that some conflicts between aspects of the system under development can only be discovered in the process of integrating different aspects of the system. The concept of collaborative definitive modelling in the DMF provides an effective way to integrate different aspects of the system, to discover and resolve hidden conflicts, and thereby helps developers to maintain the conceptual integrity of the whole system design.

5.5 Enabling technologies for maintaining conceptual integrity

EM can be viewed as an enabling technology that helps to maintain conceptual integrity in the design of systems. It uses observations, dependencies and agencies as abstractions that can potentially bring unity to diverse human interpretations and viewpoints. Where software systems are concerned, the most closely related technologies are associated with the search for paradigms for data modelling and programming that are well-matched to human cognitive processes and everyday interaction. In this section, we review key data modelling and programming paradigms, and briefly consider their connection with EM.

5.5.1 Data modelling

In the early years of computing, all persistent state was stored in program and data files. The data in such files was recorded in many different formats, using data types specific to the programs used to process the data. As applications became more complex, the diversity of data representations and structures made an integrated data processing strategy problematic. The lack of conceptual integrity in data models lay at the root of this problem. The introduction of the first databases was associated with new techniques for data modelling aimed at maintaining conceptual integrity.

The central idea behind databases is to separate the presentation of data from how the data is stored at the physical level [Car95]. This is intended to achieve machine-independent representation of data. Database users do not have to worry about how the data is stored on the physical media before manipulating the contents of a database. Early database technology included hierarchical and network databases. These were then replaced by relational databases. Relational databases succeeded because of their foundation in the mathematical theory of relations, and because their query languages allowed even end-users to manipulate the database easily. At that time, by modern standards, the applications of computers were rather limited. Relational databases helped users to maintain the conceptual integrity of systems by providing a unified way to represent and manipulate data.

In recent years, as computers have become pervasive and popular, the applications of computer-based technology seem to have outgrown the set of

primitive and predefined uses that relational databases support. Modes of data representation and manipulation are now far more diverse and complex, as are the interfaces between data, the users and the environment.

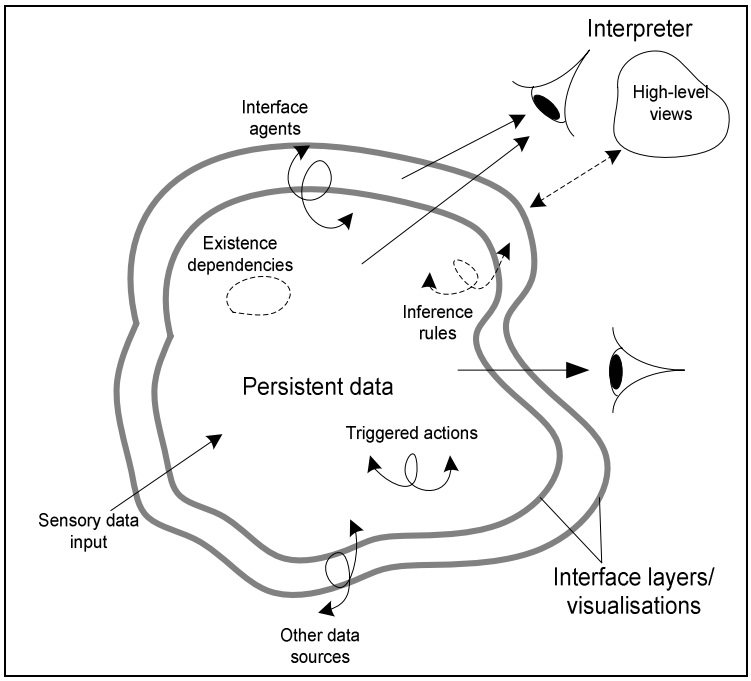

Figure 5.7: Modern context for data generation, access and manipulation

Figure 5.7 depicts the modes of data generation, access and manipulation represented in a typical modern computing application. The interactions between the persistent data depicted in the middle of the figure and the various different agents are complex. Many different forms of data input and sources of data are represented in the applications. The multi-user environment urges the consideration of different visualisations and customisations. The manipulation of persistent data is in part automated by transient processes that are hidden from the user using triggers, inference rules and data dependencies. It is then very difficult for an individual to comprehend the whole system as a whole entity. The conceptual integrity of the system is threatened. We need a better approach to data modelling that should be sufficiently expressive to meet the challenge of representing such an application in an intelligible way.

The principles of EM potentially offer an approach to data modelling that can bring conceptual integrity to modern computing applications. Observation, dependency and agency provide more general abstractions for data modelling than conventional data modelling paradigms can provide, and potentially enable data generation, access and manipulation to be managed in a unified framework. To help

the reader to understand what EM can offer, we briefly compare the qualities of an EM model with those of other data modelling paradigms (cf. [Bey94b]).

EM versus relational data modelling. The main similarity between EM and relational data modelling is that both use dependency relationships to organise observables. In relational database design, tables are created according to the patterns of functional dependency observed amongst sets of attributes. In an EM model, observations are also structured with reference to the functional dependencies between observables. However, in a relational database, functional dependency is only used for designing tables. When it comes to the use of a relational database, changes to the table structure are assumed to be rare. In an EM model, functional dependency is used to maintain potentially much more dynamic relationships amongst observables.

The wide acceptance and powerful influence of relational databases in business computing over the last 20 years can be attributed to its major contribution to meeting two research challenges: that of end-user programming (as addressed by the development of relational query languages) and that of representing real world state (as addressed by the pervasive representation of business data in relational tables). It is interestingly to note that, since Codd's conception of the relational model of data in 1970 [Cod70], the researches on the two themes of 'end-user programming' and 'representing real world state' have diverged, so that they are now conducted by-and-large separately. Where end-user programming is concerned, the modern emphasis is on visual programming. Where representing real world states is concerned the modern emphasis is on virtual reality (VR) and augmented reality (AR). The integration of these two research strands is no longer comprehensively supported by relational data modelling. The principles of EM potentially offer more scope for the reintegration of these two themes. Evidence to support this claim can be found in the fact that the Information Systems Base Language (ISBL [Tod76]), an early prototype for a relational query language, is essentially a definitive notation. This means that, by introducing the Eden Definitive Database Interface (Eddi) – a definitive notation based on ISBL – it is possible to subsume the functionality of a relational database within the EM tool TkEden. The emerging use of Eddi in conjunction with agent-related abstractions (e.g. as implemented by TkEden triggered actions) and definitive notations that offer more than tabular representations and a relational query interface illustrates the scope for EM to generalise relational data modelling.

EM versus rule-based data modelling. The need for data modelling to support rule-based activity underlying 'intelligent systems' motivated the deductive database as an extension of the relational database. In addition to all the relational database features, the database model stores mechanisms for reasoning about the stored information expressed using constructs of logic programming [Ram94]. Deductive databases can perform sophisticated inferences and draw conclusions from the data stored by using a predefined set of logical rules. In EM, the possible forms of user interaction are typically subject to fewer logical constraints. Moreover, in principle, EM can represent the ideas of triggering and deduction in ways that are safer in that they give the user more conceptual control, albeit at the cost of building explicit mechanisms that are less efficient.

EM versus object-oriented data modelling. Storing data organised as objects with relevant operations is conceptually very different from storing data organised as observables within definitions. In particular, indivisible interactions between real world entities are very well represented by definitions specifying dependencies between observables. In an OO data model, the indivisibility of interactions can only be modelled by message passing among objects. Whereas dependency maintenance in EM is automatic, in an OO model the integrity of updates can only be guaranteed by explicit specification of communications between objects. In the real world, the characteristics of an entity are determined by how it is observed. How an entity is perceived and how it can be transformed is dependent what agents are present in a system. In some circumstances, this means that what constitutes an entity is very ambiguous and changes over time. An EM model facilitates this dynamic view of entities by allowing the user to change what constitutes an observation on-the-fly. In contrast, the OO paradigm, objects have fixed boundaries, and changing the boundaries is usually difficult. Moreover, many observations may be associated with a context supplied by more than one object – as when we consider the attributes of a relation. It is difficult to model this kind of observation in an OO data model. The reason for this is that objects are good at representing circumscribed patterns of observation and transformation but not the degree of interconnectedness and fuzziness that characterises change in the real-world.

Features of the EM approach to data modelling can be summarised as follows:

- An EM model has the fundamental quality of a database that data sources are conceptually integrated but offers more scope for rich representation and interaction. Definitive notations support more general observables for

the metaphorical representation of data of interest.

- Definitions provide a direct and dynamic mechanism for representing dependencies amongst observables.
- The concept of agency in EM is general enough to embrace all activities that are associated with data generation, access and manipulation.
- EM supports the integrated design and use of data modelling applications within the same framework (cf. the 'factory' analogy in section 4.4.3).

5.5.2 Programming paradigms

In this subsection, we shall consider the efforts made in computer science to help developers to maintain conceptual integrity in software systems. The discussion will centre around how programming research is converging towards a generic modelling concept that helps to achieve conceptual integrity.

Almost every paradigm of programming represents a shift in perspective on how we do programming. Most of the well-developed paradigms have a relatively small and coherent set of concepts. For example, logic programming sees everything in terms of logic rules; functional programming sees everything as functions; and in OO programming we are encouraged to see things in object terms.

The variety of programming paradigms reflects the nature of computer programming. Software system development resembles general problem solving in that there is more than one way of solving a problem. But why is one programming paradigm more popular than another if they all seem to have a set of coherent concepts? In particular, why is the OO paradigm popular but not the functional programming paradigm? One possible explanation is that the OO paradigm is more in line with the way we think about problems. Identifying objects is one of the tasks we usually do when solving problems in the real world. We also invoke the concept of inheritance when we recognise one problem as a special case of another. The success of the OO paradigm reflects a deeper concern that relates to conceptual integrity – we are not only seeking to design a coherent language around simple and consistent programming constructs, but also a coherent as well as a commonsense way of thinking about the world. True conceptual integrity of software systems can only be achieved by finding ways to represent our natural way of thinking about the world. We can find more evidence in support of this claim by considering the extensions of the OO paradigm discussed below.

One criticism of OO concerns the hierarchical structure of class inheritance. A class inheritance diagram depicts a relationship among objects in the real world. It is a mechanism for classification. As a commonsense analysis shows, a set of concepts can usually be classified in different ways in the real world, but class inheritance only represents one of the classifications. Real classification is dynamic whereas class inheritance in OO is static. This leads people to argue that OO cannot capture real world concepts faithfully [Jac02]. Recent research into 'subject-oriented' [Har95] and 'aspect-oriented' [Elr01] programming are trying to address this problem. The term 'cross-cut' is used to describe situations in which there are common concerns amongst a set of classes that cannot be captured by a class inheritance diagram.

Other critics are challenging the concept of object itself. Agent-oriented programming can be viewed as an extension to OO. In this context, Jennings [Jen00] cites the following definition: "An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.". This area of research recognises that there are in general two kinds of object: passive and active. The current OO paradigm favours the modelling of passive objects: most objects respond to requests by means of message passing and cannot initiate actions on their own behalf. The identification of passive and active objects is also a commonsense feature of everyday life situations.

OO and its extensions reflect a trend in programming research towards finding ways of representing real world concepts as naturally and faithfully as possible. In shifting the focus from objects to agents, we are moving towards a more 'commonsense' kind of modelling. If we regard Agent-oriented programming as a 'natural extension' of OO, then EM can be thought of as a 'natural extension' of Agent-oriented programming [2]. In fact, some researchers in Agent-oriented programming are proposing concepts that are in line with the principles of EM. This will be illustrated in the following discussion, in which we compare EM with other approaches to programming.

[2] Roughly speaking, classes in OO can be represented by EM agents; inheritance in OO corresponds to prototype inheritance in EM.

EM versus object-oriented paradigm

Typically, OO software development creates three artefacts: the object-oriented analysis (OOA) model – the model of the real-world problem; the object-oriented design (OOD) model – the model of a software solution of the problem; and the program – the implementation of the OOD model. Although all three models are represented in terms of objects and their relationships, they are inherently different. As Kaindl [Kai99] observes, the transitions from model to model are not smooth. Although the graphical notations used to represent OOA and OOD models are very similar, and this gives a feeling of smooth transition, in reality it causes more confusion. Also, in practice, the transitions between different OO models are very *ad hoc* (see Figure 5.8) – updating a particular model usually triggers a chain reaction of changes in the other two models resulting in time-consuming consistency maintenance and potentially more proneness to errors. In EM, we typically have only one model to deal with. This is because, in practical situations, we do system analysis, design and implementation incrementally in parallel. The fact that there is no clear distinction between different models from different kinds of software engineering activities helps to make the system design more adaptable and eliminates potentially time-consuming transitions between different models.

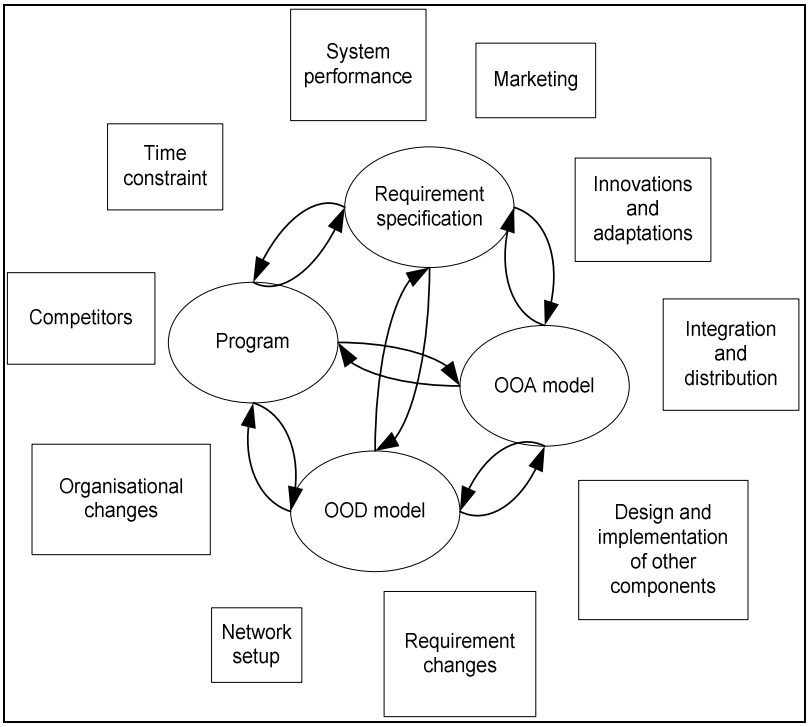

Figure 5.8: Chaotic transitions among different models in an OO software project

Another comparison can be made between an EM model and an object model of a real world entity such as a house. If we view each transformation of a definitive script to represent a house as a method, we can regard an EM model of a house as an object. This interpretation is acceptable if we already know the transformations to which a house is to be subjected in our proposed system. The distinction here is between the circumscribed knowledge required to specify an object in OO and the provisional knowledge of attributes and interactions associated with an EM model. Comprehending an object involves knowing everything we can do with it, but in itself an EM model does not circumscribe the transformations we can apply.

The difference between an EM agent and an object also reflects the different philosophies behind the two paradigms. An object contains state of its own. Communications between objects are via message passing (as implemented for instance by using synchronous function calls). Real world entities communicate with each other concurrently and freely in ways that cannot be captured by predefined message calls. EM agents can model real world entities more accurately than objects. An EM agent is more general than an object. All objects can be interpreted as EM agents but not *vice versa*.

EM versus other extensions of object-oriented paradigms

Agent-oriented software engineering can be viewed as an extension of the OO paradigm. There are two common notions of ‘agent’ in the field. The *strong* notion of agency models an agent in terms of mental notions such as beliefs, desires and intentions to be explicitly specified in both design and implementation; the *weak* notion of agency models an agent in terms of its observable properties as anything that exhibits autonomy, reactivity, pro-activity and social ability [Woo95].

If we adopt the weak notion of agency, it is difficult to identify agents because characteristics such as ‘social ability’ are difficult to define exactly. Lind [Lin00] argues for a less restrictive notion of agency. He suggests a *very weak* notion of agency whereby any entity can be an agent, and contends that “the conceptual integrity that is achieved by viewing every intentional entity in the system as an agent leads to a much clearer system design and it circumvents the problems to decide whether a particular entity is an agent or not”. This very weak notion of agency is in keeping with the EM notion of agency as discussed in chapter 2. However, the most distinctive feature of agency in EM is the mediation of agent interaction through the explicit representation of dependencies, which have so far not been well-explored in

the traditional agent-oriented paradigms.

5.6 Summary

This chapter discusses the importance of maintaining conceptual integrity in a system design as the core preliminary activity prior to system conception. We point out that the main obstacle to maintaining conceptual integrity is the representation and management of the potentially incoherent experience involved in distinguishing and synthesising viewpoints at the personal and interpersonal levels. We propose that, by using ISMs, system developers can represent incoherent experiences so that they can be studied and shared through observation, interaction and experiment. The abstract discussion of these principles is illustrated with reference to the construction of the Railway model. We have also discussed and compared EM with data modelling and programming paradigms as enabling technologies to support the maintenance of conceptual integrity in a system design.