Chapter 3

Distinguishing Empirical Modelling from programming

The introduction of EM as an approach to constructing computer-based construals as given in Chapter 2 invites a comparison with programming. A naive view would be that EM is 'just another programming technique', but critical tensions between EM and programming demonstrate that this view is incorrect. This chapter differentiates EM from programming on a fundamental level in order to show that EM offers a completely new approach to constructing computer-based artefacts as set out in Chapter 2. Furthermore, the differences stand out when considering the design and use of educational technology. From a conventional programming perspective, there are typical roles for *student*, *teacher* and *developer* for using, specifying and implementing respectively. In EM, these roles are blended because all interaction is of the same essence. EM activity is more like the use of spreadsheets as discussed in the second half of the chapter.

3.1 Five points of contrast with programming

EM has been developed over a number of years mainly by computer scientists. The tools have similarities with programming tools, and some computer scientists have used the EM tools for programming-like activities. However, the experimental, flexible and meaningful characteristics of EM suggest that it is different from programming. This section highlights five points of contrast, summarised as:

- Programs are more constrained than EM construals;
- Programming entails many discrete phases;
- Programming is concerned with developing an end product;

- Programming is concerned with the correctness of a program;
- Programming makes a distinction between development and use.

To understand the fundamental difference between EM and programming, it is helpful to compare it to the difference between radical design and routine design in engineering [Vin93]. Michael Jackson (of Jackson Software Development methods fame) argues that programming is often treated as routine or normal design [Jac06] in which "the engineer knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task" [Vin93]. In radical design, by contrast, "how the device should be arranged or even how it works is largely unknown. The designer has never seen such a device before and has no presumption of success. The problem is to design something that will function well enough to warrant further development" [Vin93]. Jackson views conventional software development methods as reflecting routine design because they treat the specification as a solid interpretation of the world, and therefore the design process is concerned with a reduced problem of how to turn the specification into a program [Jac06]. The right-hand side of Figure 3.1 illustrates routine design with respect to programming. Jackson believes that radical design cannot be fully be addressed through the concepts of routine design [Jac06]. EM can be considered more like radical design because it is concerned with interaction that negotiates meaning between a situation in the world and a construal, and with interaction that is prior to ritualisation as described in Chapter 2 and illustrated in Figure 2.1 on page 33. EM is closer to 'immature' design where the situation is not well understood, where there is no presumption of success and where the emphasis is on creating something that might stimulate further exploration. The left-hand side of Figure 3.1 illustrates radical design from an EM perspective. Figure 3.1 depicts the fluid interpretation between the world and the construal that is characteristic of early interactions prior to ritualisation. Figure 3.1 also reflects the possibility of many different outcomes from creating a construal. For example, there have been many resulting models from the 3D room model described in Chapter 4: an instrument for a teacher to give presentations relating to 3D graphics; a walkthrough of interactions on a 3D room for a student to follow; and, an exploration environment for students learn about 3D issues in computer graphics.

The contrast between radical and routine design is relevant to the following five sections which elaborate on the five distinctions outlined above between EM and programming.

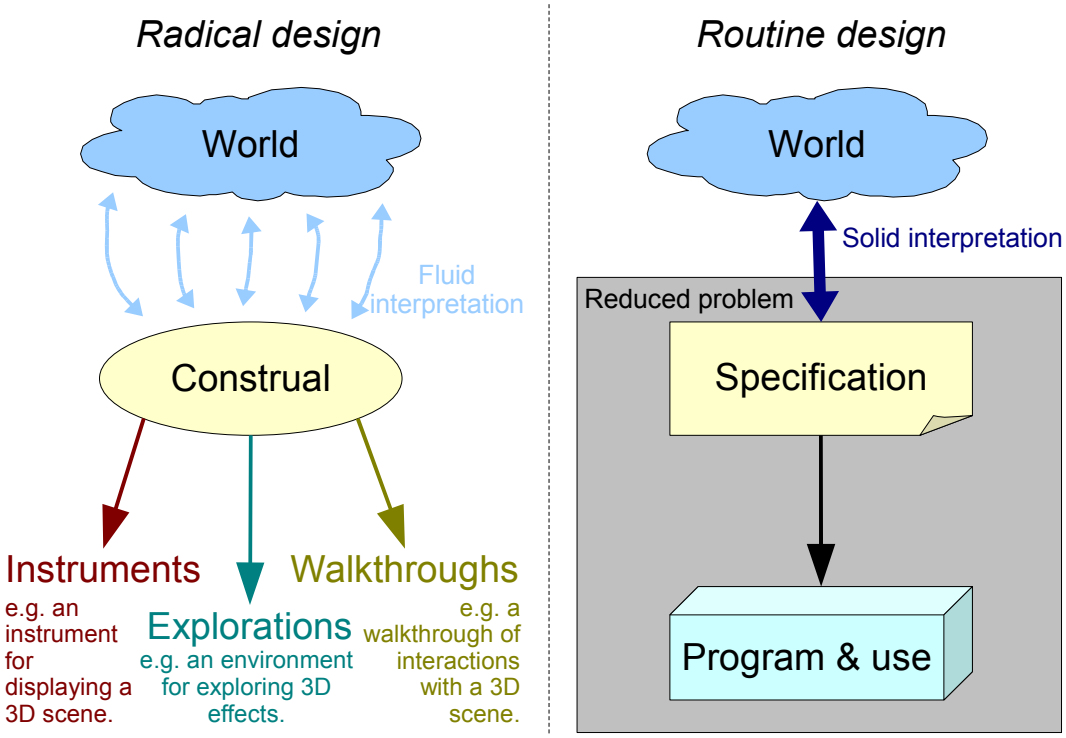

Figure 3.1: Contrasting EM and programming from a radical and routine design perspective.

3.1.1 Programs are more constrained than EM construals

The discussion in Chapter 2 associates EM activity with the construction of construals that reflect everyday concurrency in a situation or referent. The goal of such activity is to 'make sense of', realise, and understand the nature of some aspect of the referent or the referent itself. The activity is concerned with experience that comes before a formalised understanding or that is *prior to ritualisation*—see Figure 2.1 on page 33. A model-builder interacting with an EM construal is exposed to experiences that can be unpredictable in a fresh or raw sense, and a model-builder typically does not know if such interactions can be repeated or ritualised. Programming is more concerned with creating artefacts with predictable behaviours: a program is a ritualisation of a specific function that the user can repeat over and over in a predictable manner.

In Rungrattanaubol's *A treatise on modelling with definitive scripts*, it states that EM activity—referred to as "modelling with definitive scripts"—is not like programming [Run02]. She explains that programming "focuses on the representation of actions and behaviours", whereas modelling with definitive scripts "is more closely related to building a physical artefact" [Run02]. Programs are designed with specific functions in mind whereby the program reflects a closed-world model of the requirements. It is a closed-world model in the sense that it encompasses the actions and behaviours that the programmer chooses

to implement. Such programs are likely to have certain characteristics, as described by Rungrattanaubol: specific modes of use, standard patterns of behaviour, standard user interaction patterns, standard interpretations of state change, clear identifying boundaries. In contrast, Rungrattanaubol observes that EM construals are open-ended models that do not necessarily have such constrained use [Run02].

Programs are closed-world models of specific ritualised behaviours due to the solid interpretation of the relationship between the program and the situation in the world as shown in Figure 3.1. The interpretation is solid in that the program has a single clear correspondence to the world defined by its specification. EM, on the other hand, is about developing a correspondence between the construal and the world—a correspondence which might change and grow stronger as the construal is constructed. EM construals are open-ended models that have a very fluid interpretation of the world. It is therefore important to distinguish the constrained nature of programs from the open-ended nature of construals.

3.1.2 Programming entails many discrete phases

EM is essentially an activity that involves interaction with a state to develop a correspondence between a construal and a situation in the world. The interaction through model-building involves a blended mix of creation, experimentation, observation, adaptation, exploration and manipulation—all of which are state-changes. Programming activity, in contrast, involves many discrete phases of different character as shown on the right-hand side of Figure 3.2. The traditional view of software development is captured in the systems life cycle as explained in Pressman's book on *Software Engineering* [Pre05]. This is a structured sequential process as shown on the left-hand side of Figure 3.2, starting with requirements, moving through specification, design, implementation, and ending with testing and integration. Although the systems life cycle model has been highly criticised, its successors retain similar discrete phases for requirements, specification, design and implementation. Newer, more popular, methods generally involve the iteration or repetition of similar discrete phases (e.g. rapid prototyping involves iteratively specifying and implementing, and eXtreme Programming repeats phases such as coding, testing, listening and designing [Bec00]). As yet, there is little evidence of any fundamentally new methods that could support the fluidity of radical design.

Recent work by Beynon, Boyatt & Russ shows that EM necessitates a radical rethink of programming whereby there is no distinction between the stages of specification, design

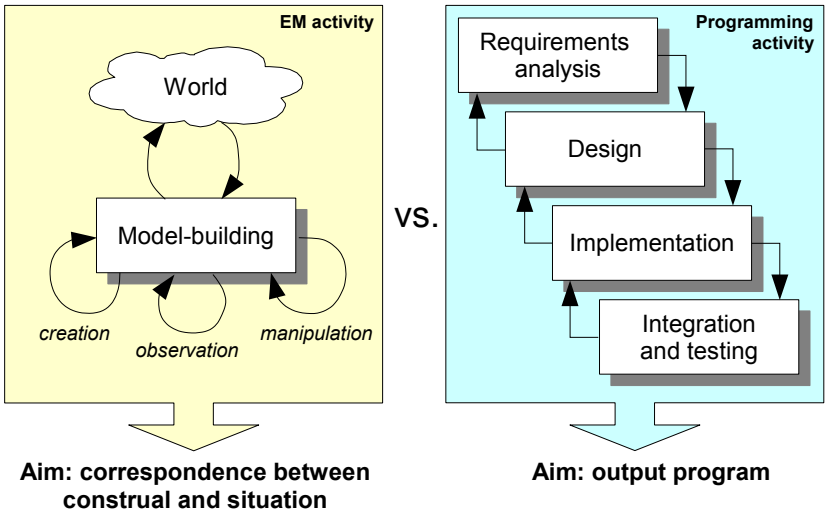

Figure 3.2: The discrete phases of programming contrasted with EM.

and implementation [BBR06]. Such ideas are reflected in programming practice as discussed by Weinberg, who states that "[in most cases] we do not know what we want to do until we have taken a flying leap at programming it." [Wei88:p12]. Weinberg goes further to undermine the discrete stages of software development: "Specifications evolve together with programs and programmers. Writing a program is a process of learning—both for the programmer and the person who commissions the program." [Wei88]. A full discussion of EM as an approach to software development is beyond the scope of this thesis (and has already been comprehensively covered by Rungrattanaubol [Run02]). The important point is that the continuous blended activities in EM are quite distinct from the discrete phases involved in programming. EM is a *holistic* approach to building computer-based artefacts that cannot simply be compared to any particular phase of programming, or the entire programming process.

3.1.3 Programming is concerned with developing an end product

Programming involves translating a specification into a program. The aim of programming is to produce the program as the end-product, as shown in Figure 3.2. In contrast, EM is not primarily concerned with producing any output. Primarily EM is about developing understanding of a situation in the world by constructing a construal. EM is not an activity which has a clear beginning or end, and construals are always subject to revision and extension. EM does not necessarily lead to a 'final' model as an end-product (see the discussion of EM characteristics in §2.2.4 & §2.2.5). In a similar manner to radical design, EM activity may result in many outcomes as illustrated in Figure 3.1, and the outcomes

are not predictable like they should be in successful routine design.

Because the aim of programming is to produce a single end-product, programming is not well matched to the aims of everyday learning which typically might involve many unpredictable outcomes. From this perspective, EM has much more in common with learning than programming because it is an activity that is well aligned to sense-making and developing understanding, and the outcomes of EM activity are not constrained. The activity of sense-making through the construction of a construal is really concerned with learning about a situation, not producing a model.

3.1.4 Programming is concerned with the correctness of a program

Given that a programmer approaches the programming task with the specification already defined, it is important that the program produced accurately represents the specification. This leads to an area of computer science known as program validation and verification. Many computer scientists are concerned with what Jackson calls 'reduced problems' [Jac05]. A reduced problem is simply turning a 'functional specification' into a program without the concerns of the situation in the world that the program relates to, as illustrated in Figure 3.1. Dijkstra, famous for his many contributions to computer science, particularly formal verification, believed that the activity of coming to a functional specification was for others to address, and that computer scientists should solely concentrate on applying formal techniques to develop programs that 'correctly' adhere to the functional specification [Dij89]. Jackson is one of a few people (supporters of eXtreme Programming included [Bec00]) who advocate programming methods that are concerned with the relationship between the world and the program/specification.

Where there is a single unambiguous interpretation of the world in the specification, and the world is therefore placed in the background, it is possible to reason about the correctness of the program in relation to the specification. However, EM *is* fundamentally concerned with negotiating the interpretation of a situation in the world with respect to the artefact. Therefore it is much harder (if possible at all) to evaluate the 'correctness' of the artefact. The fluid interpretations that are possible with a construal are open-ended, changing, and not subject to correctness concerns. If EM is like radical design, as depicted in Figure 3.1, then the concerns are simply whether a construal might stimulate further exploration and better understanding. EM is much more concerned with the usefulness, rather than correctness, of an artefact.

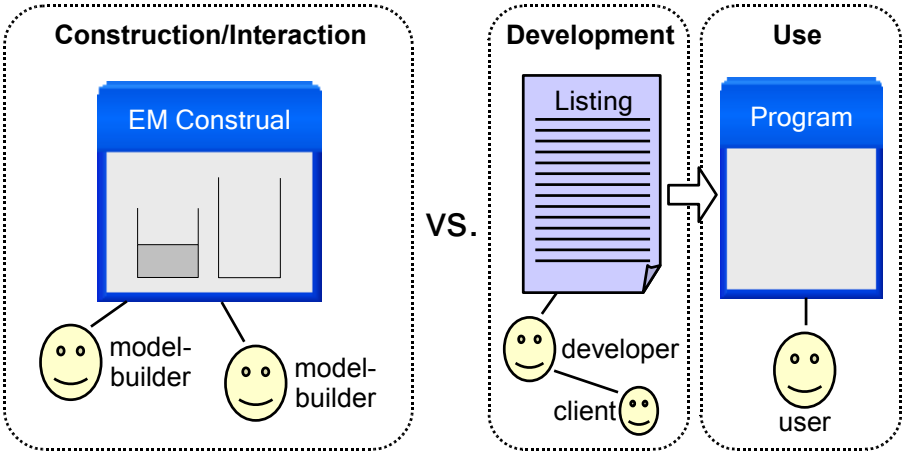

Figure 3.3: EM contrasted with conventional software development and use.

3.1.5 Programming makes a distinction between development and use

At the end of a programming activity, when the program is complete, it is delivered to the user where the program submits itself wholly to software use, as depicted on the right-hand side of Figure 3.3. The activity of programming implies a distinction between development and use. The programmer develops the program, and the user uses it. The user does not develop the program, and rarely does the programmer use the program. The software engineering process often involves a third party—the client—as shown in Figure 3.3. The client is described as the person who specifies the requirements for a new piece of software [Pre05].

In EM, development and use are combined, and can be treated as one and the same activity. The distinction between construal creation and program development/use is depicted in Figure 3.3. When the aim of the construal creation activity is developing a correspondence between the construal and the situation modelled, it is clear why constructing a model and using it are inseparable. Rungrattanaubol shows that because EM "respects the continuity of the modeller's perception, design and use can be interleaved without interference, and the role of the modeller is more aptly characterised as 'designer-user' rather than 'design-exclusively-or-user"' [Run02]. The consequences for drawing no distinction between development and use are highlighted later in the chapter when considering educational technology.

3.2 Illustrating the distinction

An example of three varieties of an educational artefact are used in this section to elaborate on the differences between software development and EM. JUGS is a simple educational

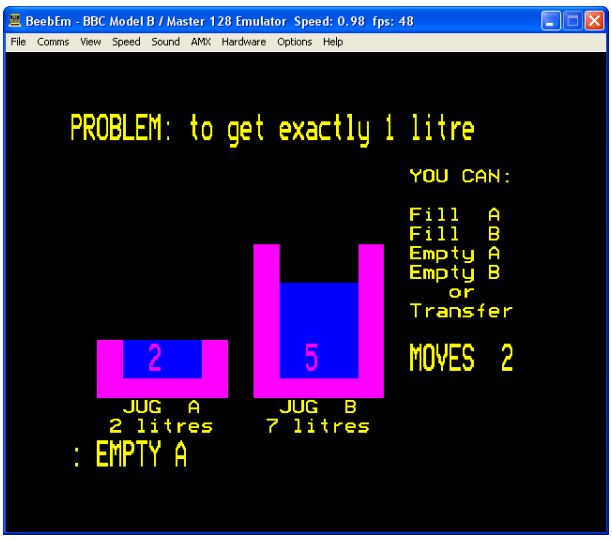

Figure 3.4: The JUGS program (running on the BeebEm emulator).

program that was initially developed by Ruth Townsend for the BBC Microcomputer in 1982 (see Figure 3.4 for a screenshot of JUGS restored with the BeebEm BBC Emulator). The underlying educational objective of the JUGS program is to familiarise school children with elementary concepts of number theory. The idea is to create an environment for exploration in which students can come to appreciate that the highest common factor of two positive integers determines what numbers can be derived by repeatedly applying addition and subtraction operations. The program presents the student with two jugs of non-identical integer capacities and options to fill or empty each jug and to pour from one jug to the other. The student is expected to apply these operations to reach a target quantity. The mathematics embodied in the use of JUGS is, given two integers `m` and `n`, the set `AS(m,n)` of numbers that can be generated from `m` and `n` by additions and subtractions alone is the set of multiples of `hcf(m,n)`. In applying fill, empty & pour operations to two jugs that have integer capacities, it follows that every operation generates a quantity of liquid that is in `AS(m,n)` and that this quantity is restricted to be positive and cannot exceed `max(m,n)`. It is also true that all quantities satisfying these constraints can be derived in this way.

3.2.1 Developing a JUGS program

Some insight into the development of the JUGS program can be gained from analysing the source code. The program is 450 lines of BBC BASIC code containing 36 procedures. There are global variables for the capacity (`capA` and `capB`) and the level of the liquid (`levelA` and `levelB`) of each jug and for the target quantity of liquid (`target`). The code

that assigns and uses these 5 variables is a tiny proportion of the entire program (only 44 lines)—less than 10%. There is a procedure for demonstrating how the fill, empty & pour operations can be used to reach a specific target which uses an algorithm to obtain the highest common factor. The code that implements the relevant mathematics for this is only 11 lines long. The majority (at least 51% or 229 lines) of the code is concerned with drawing or printing to the screen. Another proportion (19% or 86 lines) is concerned with the input and decoding of commands. Figure 3.5 shows the source code together with the control flow of the program, with the 5 variables relevant to the task in bold. Many of the connecting lines have a relationship to dependency maintenance in that the procedures are updating part of the state, particularly the screen state. For example, whenever the level of either jug changes, the procedure `PROCdisplayupdate` is called to update the jugs on the screen in order that changes to the `levelA` and `levelB` variables can be observed.

The JUGS program may seem small and trivial by today's standards, but it illustrates that even the development of simple programs requires a considerable knowledge of a procedural language. Less than 10% of the code in the JUGS program is actually in the subject area of jugs, and even less of it relates to the highest common factor calculation to find out the potential target quantities. This provides evidence that much of the development activity for the JUGS program was not in the domain of mathematics and jugs. Given the large proportion of code relating to input (decoding commands) and output (printing to the display), it would appear that the majority of development effort went into areas unrelated to mathematics and jugs. Observations made by Weinberg indicate that successful programmers are often those who are willing to spend long periods of time puzzling over the intricacies of a program [Wei88]. From a constructionist learning point of view, the development of the interface is probably not the most important activity, especially if it is divorced from the activity of understanding the issues surrounding the highest common factor.

3.2.2 An EM approach to JUGS

The EM jugs model was originally developed by Beynon as an example of software development using a dependency-based environment [BNR89] [EMP:jugsBeynon1988]. It has been reused and extended for a number of subsequent modelling exercises. Rungrattanaubol improved the realism of the jugs model by modelling the liquid at a pixel level and adding other observables that effect the evaporation of the liquid [EMP:jugsextensionsRun-

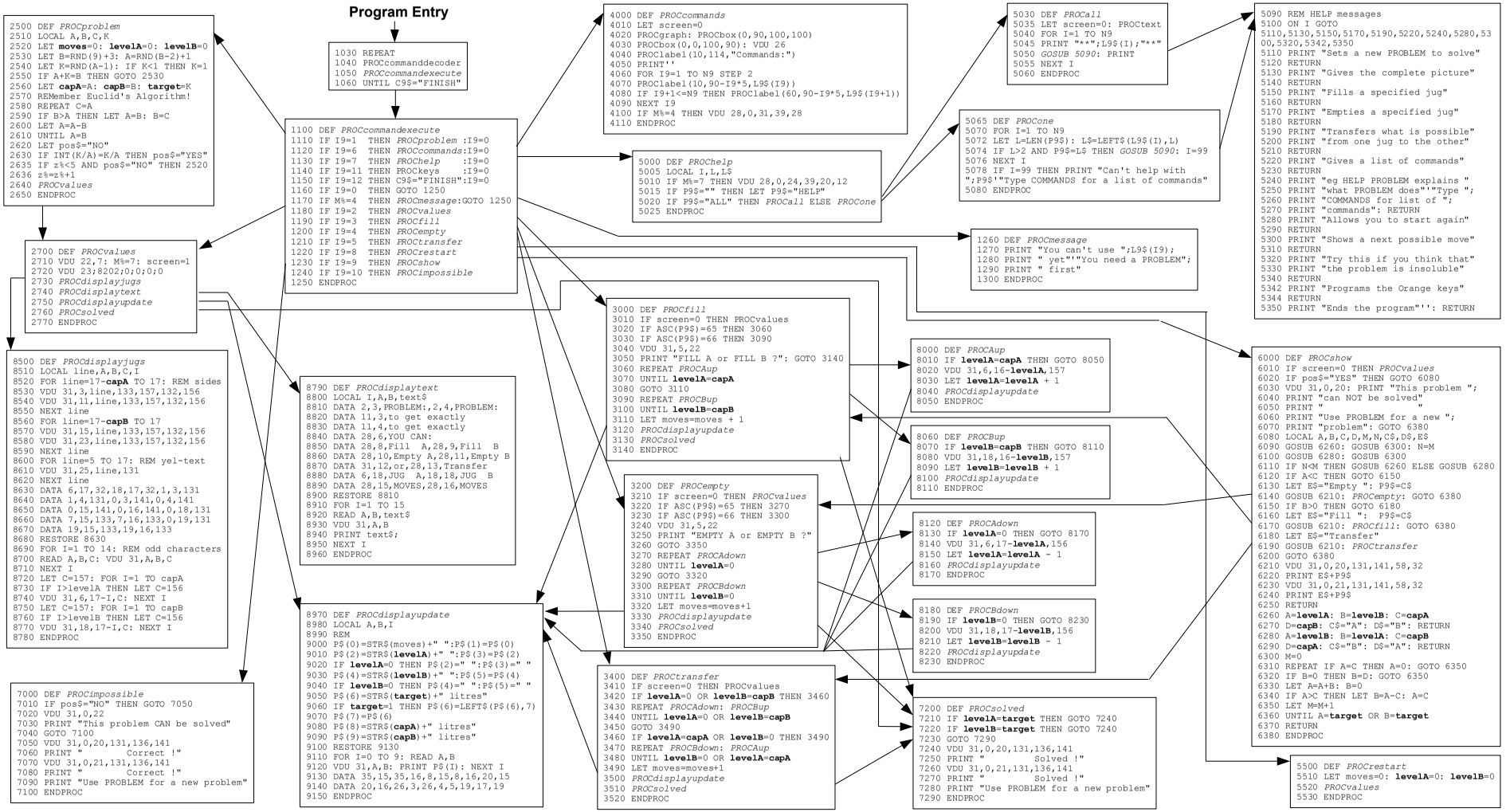

Figure 3.5: The listing and flow of the JUGS program.

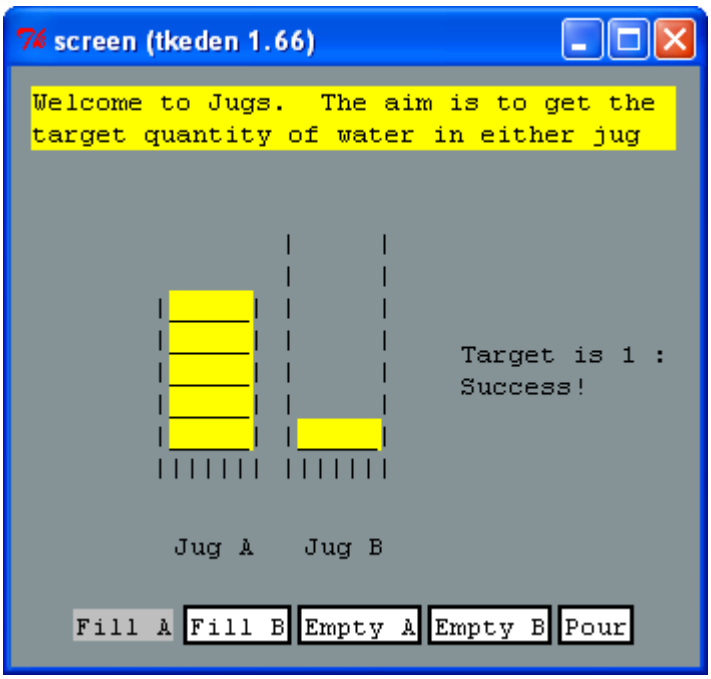

Figure 3.6: Pavelin's variant of the jugs model [EMP:jugsPavelin2002].

bol2002]. Beynon extended the jugs model by interpreting the jugs as strings and frets on a guitar, as discussed by King [Kin07]. During a final year undergraduate project, Reynolds morphed the jugs model into a model of a bar where the contents of each jug represented a queue of people [Rey04]. The jugs model has also been used extensively for teaching and for laboratory sessions in an introductory EM module on the Computer Science course at Warwick (as discussed in Chapter 6). Pavelin's version of the jugs model [EMP:jugsPavelin2002] (see Figure 3.6) and King's introductory presentation using jugs [Kin07] are commonly used as models for introducing EM principles and practice.

Although the jugs model is a simple example of EM, comparing it with the original JUGS program highlights a number of important differences. Figure 3.7 shows the jugs model and a definitive script that embodies the part of the jugs model relating to the status bar. During interaction with the jugs, the contents of the status bar corresponds to an aspect of the current state. For example, if either of the jugs is currently filling, emptying or pouring, then the status bar will show "updating". Both the screen state and the underlying state are maintained by dependency. The current state in Figure 3.7 is that the target of 1 unit of liquid has been met in Jug B. In other words, `contentA` is equal to `target`. Through dependency, the observable `finish` is true, as long as `contentB==target` remains true and assuming `active` is false. Once again through dependency the observables `update_status`, `status` and finally `status_box` are maintained in response to changes to `finish` and other observables. The `status_box` definition is defined in the SCOUT nota-

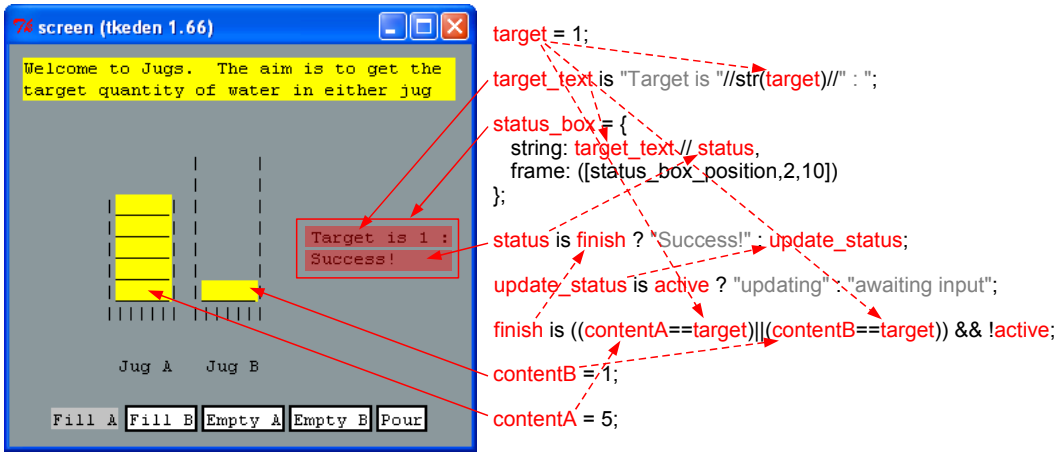

Figure 3.7: The jugs model and a part of the script relating to the status bar.

tion for screen layout, and embodies the status bar artefact displayed on the jugs screen as highlighted by the red outline. The two red shaded areas represent two parts of the `status_box` string, the first dependent on the `target_text` and the second dependent on the `status`. The benefit of using dependency is evident when comparing the script in Figure 3.7 with the part of the program listings in Figure 3.5 involving `PROCsolved` (second from the right at the bottom). The EM jugs model shows "Success!" in the status bar by dependency whenever the selected observables are in the described state. With the JUGS program, the programmer must manually run a checking and updating procedure whenever he deems the selected observables might have changed. Although writing a procedure to update the status bar may be fairly trivial (such as in `PROCsolved`), reasoning about all the places where the procedure should be called from is more difficult. Furthermore, if a program changes over time, by adding another jug for instance, it becomes even harder to ascertain whether the "Success!" in the status bar will always correspond to realising the target. An issue was discovered in the original JUGS program relating to this update problem: when the target is reached "Success!" is displayed as expected, but if you then make an error, or request help, then "Success!" disappears. Such problems are rarely a concern for a model-builder using dependency.

If the model is interacted with in a restricted way (i.e. using the interface buttons alone) then, as Rungrattanaubol points out, it serves the same function as the JUGS program [Run02]. However, the jugs model can be interacted with in an open-ended manner that has no counterpart in the JUGS program. When the JUGS program is executed, it is a closed-system and there is no potential for observing the internal state of the program or alter the state at any time during the execution (except in ways allowed

by the specification). In the EM model of jugs, the underlying state is exposed and open to change at any time during construction and use (between which no distinction is made as described in §3.1.5). The number of models that have built upon jugs demonstrates that models can be extended rapidly for a variety of purposes. Furthermore, extensions can be performed in the stream of modelling because EM construals are visible and alive, constantly running even during modification. This reveals a clear distinction between EM construals and programs.

Rungrattanaubol discusses the differences between programming and EM by talking about the explicit and internal states [Run02]. The *explicit state* is the state of the model as the model-builder observes it (i.e. graphical interface). The *internal state* is the state of the internals of the program, for example, the variables and procedures. In a procedural program the internal state is almost impossible to observe. In the JUGS program there is no direct relationship between the variable representing the capacity and the explicit drawing of the jug on screen. It is left to the programmer to decide under what circumstances the screen should be redrawn, given that the screen need not be updated in every situation. In EM, with the use of dependency and notations such as DoNaLD and SCOUT [War04], the model-builder can construct an explicit state that is linked to the internal state, and vice versa. The internal and explicit states are kept consistent at all times; a change in the internal state will be immediately reflected in the explicit state as viewed by the model-builder. In the jugs model, for example, it becomes difficult to differentiate between internal state and explicit state: is `status` explicit or internal? Internally it is a string, explicitly it is a part of the status bar that can be observed. In EM, all the internal states have the potential to be explicitly observed, hence it is not necessary to differentiate as is common in programming.

3.2.3 JUGS by object-orientation

It could be argued that our comparison with a program written in BBC Basic is unfair, because nowadays object-oriented (OO) languages are more commonplace and this makes programming easier or removes some of the problems of procedural programming. From a novice perspective there are difficulties associated with learning OO languages when compared to high-level procedural languages such as BBC Basic. During the time of the BBC Micro, teachers wrote their own small, highly specialised programs for their students. How often does a teacher write a Java program for their students? Educational

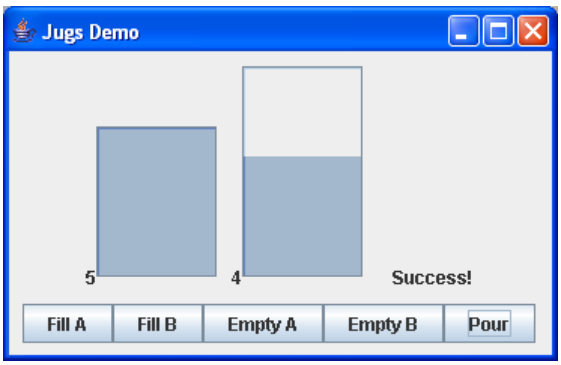

Figure 3.8: The Java JUGS program.

technology is usually built by professional programmers and it is often general purpose software designed for a wide range of teaching tasks (e.g. Toontalk [Kah96], Imagine Logo [KB00]). Programming in Java in many ways is more difficult than the days of the BBC Micro. One only has to compare the number of lines required to write a "hello world" program in Java to a similar program in Basic to see that OO programming does not always make programming easier for novices. Sheetz et al [She97] explore in detail the difficulties of OO design and programming. They found that learning the basic concept of objects was generally difficult, even before moving on to more complicated issues such as class reuse. Wiedenbeck et al [WRSC99] discovered in experiments that procedural programming students' were superior to OO programming students' in the comprehension of large programs in every category analysed. The findings suggest that although OO programming is a successful paradigm for software development, it is not necessarily an easy approach to programming for novices.

A Java version of the JUGS program was implemented by the author to demonstrate the differences when compared with the Basic JUGS program and the jugs model. The development of Java JUGS was undertaken in one morning, and it contains only very basic functionality compared to the Basic JUGS program. As shown in Figure 3.8, there are two jugs displayed and 5 buttons for filling, emptying and pouring.

Java JUGS is only 222 lines long split across two classes. The use of classes can be seen as an advantage over the original JUGS program because Jug is a class and so therefore creating two or more jugs is a simple matter of declaring and instantiating each Jug object. I was also able to make use of the standard GUI (Swing) components and layouts to arrange the buttons and jugs (as opposed to the primitive lines and text in BBC Basic). The jugs shown in Figure 3.8 were in fact standard JProgressBar objects (oriented vertically) that had a specific size (the height of the jug) and their progress

value represented the level of the liquid in the jug. The use of pre-built components can speed up development time and reduce the amount of code required to perform input and output. It also ensures that Java JUGS fits in with the operating system (in this case Windows) and looks and feels like a standard application.

Although the use of standard GUI components simplifies the input–through pre-built buttons–and the output–through progress bars and labels–a large proportion of code in Java JUGS is still dedicated to presentation. At least one third of the code (80 lines) is dedicated to building the graphical interface, without including the code that is required to keep it up-to-date during interaction. An extract from the main Java JUGS class is shown in Figure 3.9 that demonstrates the nature of this code. In comparing this code to that of the BBC Basic JUGS, the latter is concerned with constructing the interface by lines and strings, whereas the former is constructing the interface by buttons, labels and progress bars. The objects may be different, but the task is the same, and the difficulty of updating the interface and maintaining consistency in Basic JUGS is inherited in Java JUGS. From an EM standpoint, these difficulties come from the apparent division of the interface from the underlying core of the program. In Java JUGS, the interface is quite independent of the Jug class, encouraging the development of an underlying abstract model that is separate from the interface through which interactions with the jug are performed. In the EM model, dependency ensures that the interface is the model. The separation between interface (the explicit state) and underlying operation (the internal state) is blurred, usually to the extent that model-builders do not talk about 'the interface'. Whereas a program can consist of an interface and an underlying abstract or mathematical model, an EM artefact creates no separation and therefore in referring to a model it implies the whole artefact (not some abstract/mathematical model underneath the program).

Even if we agree with Meyer's contentious claim that "object-oriented designers usually do not have to spend their time in academic discussions of methods to find the objects: in the physical or abstract reality being modelled, the objects are just there for the picking!" [Jac05:p14], object-orientation can lead to other problems. Information hiding as a feature of object-orientation has its advantages, but deciding *what to hide* in classes can be troublesome. There are some objects or methods that might clearly belong to the Jug class, but there are others where classification is not so trivial. In experiments with microworlds, Goldstein et al. [GKN01] observed a number children grappling with similar concerns as to whether to put the X in the Y or the Z. As a personal example, during

```
 92  private void buildGUI() {
 93     JFrame frame = new Jframe("Java JUGS Demo");
 94     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 95     JPanel panel = new JPanel(new BorderLayout());
 96     frame.getContentPane().add(panel);
 97
 98     panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
 99
100     //Add labels for jug content/level.
101     levelAlbl = new JLabel("0");
102     levelBlbl = new JLabel("0");
103
104     //Add progress bars to represent jugs
105     levelApb = new JProgressBar(SwingConstants.VERTICAL,0,jugA.getCap());
106     levelBpb = new JProgressBar(SwingConstants.VERTICAL,0,jugB.getCap());
107     levelApb.setMaximumSize(new Dimension(80,20*jugA.getCap()));
108     levelApb.setMinimumSize(new Dimension(80,20*jugA.getCap()));
109     levelBpb.setMaximumSize(new Dimension(80,20*jugB.getCap()));
110     levelBpb.setMinimumSize(new Dimension(80,20*jugB.getCap()));
111     levelApb.setAlignmentY(Component.BOTTOM_ALIGNMENT);
112     levelBpb.setAlignmentY(Component.BOTTOM_ALIGNMENT);
113
114     //Add panel for jugs
115     JPanel jugpanel = new JPanel();
116     jugpanel.setLayout(new BoxLayout(jugpanel, BoxLayout.X_AXIS));
117     jugpanel.add(levelAlbl);
118     jugpanel.add(levelApb);
119     jugpanel.add(Box.createRigidArea(new Dimension(10,0)));
120     jugpanel.add(levelBlbl);
121     jugpanel.add(levelBpb);
122     jugpanel.add(Box.createRigidArea(new Dimension(10,0)));
123     panel.add(Box.createRigidArea(new Dimension(0,10)));
124     panel.add(jugpanel);
```

Figure 3.9: An extract from Demo.java that constructs the GUI for the Java JUGS program.

development I was unsure whether to put the 'fill' and 'empty' methods inside the Jug class. The reason I did not was because these two operations are similar to the 'pour' operation which has to work from one jug to another, and so I placed them all together with the button functionality. The emphasis on objects in Java JUGS leaves question marks around where to put the agency: is it in the object's methods, or is it in the button's action? Another difficulty I came across when deciding what to put in the class was with the interface components. My Jug class described the abstract properties of the jug, i.e. its capacity and its current level of liquid. When I constructed the user interface I created two progress bar objects for the graphical display of their capacity and content. After this I struggled with my reasoning as to the 'correct' place for these components. Should the progress bar be placed inside the Jug class or should it be placed with the other GUI components? Initially I decided the jug interface components should be put inside the jug class, but then after further deliberation I realised that this might severely limit further use of the jugs if it were to be morphed into another application such as King's guitar strings [Kin07]. A more experienced object-oriented programmer would no doubt spend less time picking the relevant bits for a class, but still there is the issue, as there is with traditional programming, that making decisions about the structure of a program may restrict future developments—which is not desirable for learning in a constructionist sense.

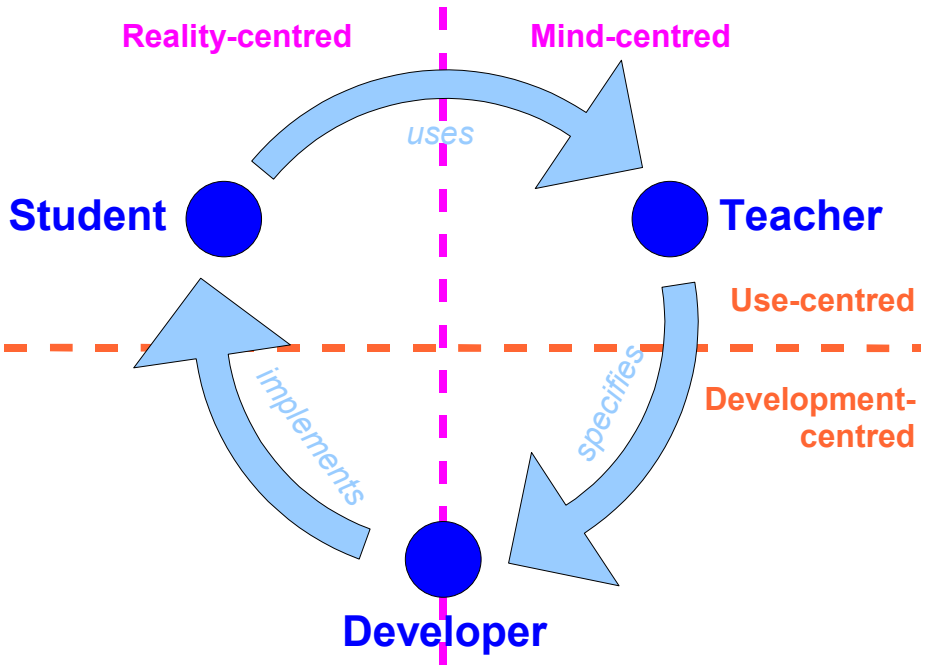

Figure 3.10: A typical software development view of educational technology displaying the perspectives of student, teacher and developer.

3.3 Implications for educational technology

Educational technology generally follows a pattern of development and use that mimics traditional software where the *student* can be thought of as the user, the *teacher* as the client, and the *developer* remains the same. The activity depicted in Figure 3.10 is a typical software development view of educational technology. The cycle can be seen as starting with the teacher who *specifies* requirements for a computer-based tool for supporting student learning. The developer takes the specification and *implements* a solution (e.g. a program or group of programs). A number of students can then *use* the tool to support or supplement their learning activity.

Each of the three roles are associated with distinct activities, typically: specification, implementation and use. Furthermore, the concerns in these activities create two important tensions. The first tension is that the developer is focussed on development, whereas the student and teacher are focussed on use. The horizontal line in Figure 3.10 signifies the division of software development versus software use. The second tension is that the teacher is concerned with teaching particular concepts (public knowledge in the lower realm of the EFL), but the student is relating to the technology and their experience of it (practical knowledge in the upper realm of the EFL). Therefore the activity of the teacher is mind-centred and the activity of the student is reality-centred, as depicted in by the

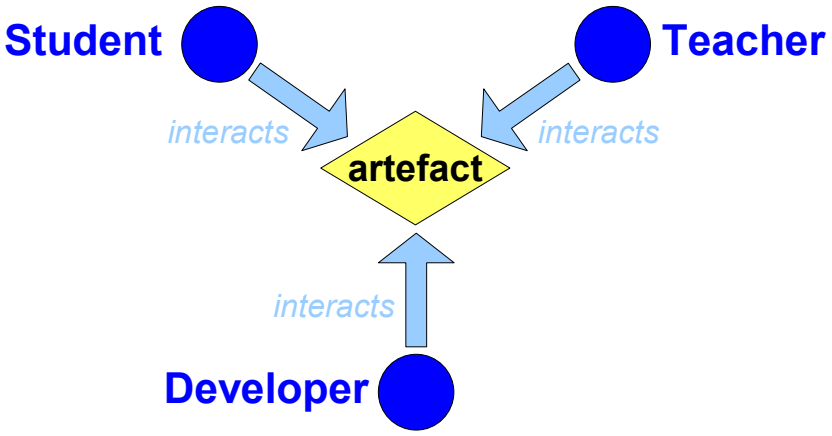

Figure 3.11: An EM view of educational technology where student, teacher and developer can be viewed as performing comparable interactions with an artefact.

vertical line in Figure 3.10.

EM is primitively concerned with modelling state-as-experienced, and all interactions with an EM artefact are state-changes. The activities of the student, teacher and developer are comparable because state-change occurs only through re-definitions. A student exercising the artefact is changing the state, a teacher preparing the artefact for a particular exercise is changing the state, and a developer constructing the artefact is changing the state. In other words, at a primitive level all possible interactions with the artefact are similar activities as in Figure 3.11. This is not to say that all interactions have to be at the level of making a single redefinition to an observable through the input window, an interaction such as a button press can be linked—through dependency or agent actions—to a series of redefinitions or state-changes in the artefact.

Following the EM approach, the implications for learning are that there is more potential for learners to interact with an artefact in a wide range of activities that would normally only be associated with either students, teachers or developers. A student playing with the jugs model can be a teacher by changing the initial state to make the target quantity impossible to reach, or a developer by adding a third jug for example. Furthermore, the blurring of the roles of student, teacher and developer removes the tensions between development and use, and between reality-centred activity and mind-centred activity, as found in Figure 3.10. Roe, Pratt and Jones [RPJ05], in reflecting on their experiences of building microworlds for mathematics education, highlight the need to give learners control to express their own ideas in their models. They add that web-based learning environments are primarily focussed on the production and delivery of content and offer little support for a learner to construct models [RPJ05]. EM facilitates such construction-

ist learning because there is little to distinguish between software development and use, and between the activities of students, teachers and developers.

To go one step further than Roe, EM can be distinguished from many constructionist environments (e.g. Imagine Logo [KB00], Toontalk [Kah96], and Agentsheets[Rep93])[†] because it does not support construction through programming—EM offers an alternative paradigm for construction based on observables, dependencies and agency (as described in §2.2.2). Programming as a method for learning by constructing artefacts has had some success. For example, ToonTalk is a visual environment for doing computations where each operation (such as assignment, branching, and looping) is given a visual metaphor (i.e. a loop is represented by a robot performing an operation many times) [Kah96]. Although ToonTalk wraps up the programming constructs in visually appealing ways, the conceptual difficulties of programming and object-orientation (as highlighted in §3.1) are hidden rather than resolved. Metaphors may help children to grasp the nature of the procedural programming, but they do not take away the difficulties of programming. EM is a possible solution for what is described as the need to put 'the learning back into e-learning' [RPJ05] by taking away the constraints of programming.

3.4 Spreadsheets as an illustration of modelling with dependency

Spreadsheets offer an environment quite different from programming—most notably in the way that development and use are integrated. In some respects, EM's distinction from conventional software development can be appreciated by demonstrating the similarities between EM and spreadsheets. The aim of this section is to trace the history of an application which may not have initially been conceived as a modelling environment but which is now used for a wide variety of modelling activities. Spreadsheets, as well as being related to EM, are also important applications in education. Therefore, a treatment of the topic of spreadsheets is highly relevant to strengthen the connections between EM and learning developed in this thesis.

3.4.1 Tracing the rise of the spreadsheet

Before the arrival of computers, a 'spread-sheet' was the name given to a large sheet of paper containing rows and columns that could be used for financial activities such as

[†]Roe discusses Imagine Logo [Roe03:p3] [RPJ05], Toontalk [Roe03:p4], and Agentsheets [Roe03:p36] in relation to EM.

book-keeping. When computers became commercially available, some of the first batch programs were designed to deal with payrolls and other finance-related activities. Many of these programs produced printed output in the form of large lists or tables similar to the traditional spread-sheets. It was not until Dan Bricklin came up with the idea of an interactive visible calculator, and developed the VisiCalc application in 1979 [Pow04], that computer-based interactive spreadsheets started being used [†]. In 1983, two years after the IBM PC was launched, Lotus Development Corporation released the Lotus 1-2-3 spreadsheet application for the IBM PC, which offered many benefits over VisiCalc [Pow04]. This application turned out to be the "killer-app" for the IBM PC, and played a significant role in the huge success of the IBM PC [Pow04]. In the late 1980s as Microsoft Windows took over from MS-DOS, Microsoft was quicker to develop a Windows spreadsheet application, and hence Microsoft Excel took over from Lotus 1-2-3 as the leading spreadsheet application [Pow04].

Spreadsheets have been widely used for financial and accounting purposes, but there are many other applications where they have made a significant impression (e.g. mathematics modelling, cellular automata and neural networks). Spreadsheets are used by software developers for building applications. Since Visual Basic for Applications was introduced into Microsoft Excel in 1993, developers have been using the Excel platform for small, often client-specific applications. For example, the Warwick Spreadsheet System (WSS) developed by Beare makes use of Excel and provides an environment teaching mathematical modelling that can be used in science and mathematics education [Bea96]. The merits of applying spreadsheets principles to software development have been noted by Nardi [Nar93]. Spreadsheets have also played a roll as a general paradigm for computing practice. The first integrated office application that Lotus Development Corporation released, called Symphony, was based wholly on the spreadsheet principle: "While different environments display information in slightly different ways, Symphony is always basically a spreadsheet. The Row/Column structure is there, to one degree or another, regardless of which environment you are using" [Bad85]. There is plenty of evidence to suggest that spreadsheets have much to offer for constructing and using computer-based artefacts.

The potential of spreadsheets is perhaps best summarised with a quote by Alan Kay: "A spreadsheet is a simulated pocket universe that continuously maintains its fabric; it is

[†]The first known implementation of an interactive spreadsheet program was developed on an IBM mainframe at Imperial Chemical Industries in the UK and was used as early as 1974 [Pow04].

a kit for a surprising range of applications." [Kay84]

3.4.2 Spreadsheets for learning

The JUGS program described in §3.2 is typical of ET around the time when computers were first introduced into schools. Such simple programs were often developed by teachers themselves. As software became able to perform ever more complex tasks, programming became a more specialist task. Simple programs by teachers were replaced by fully-featured educational software by teams of programmers. It was not feasible for all teachers to learn to program and to write their own programs. There was still a need though for teachers to build their own computer-based artefacts for specific topics.

Baker & Sugden, in the first article of their electronic journal *Spreadsheets in Education*, discuss how spreadsheets have been used in education since Lotus 1-2-3 was first introduced [BS03]. At this time computers were seen as having potential in an educational setting, but in most cases teachers or students had to learn a programming language. The other option was to purchase a specific piece of software, but each subject required at least one piece of software, and every piece of software had its own interface and style of use. However, this changed when spreadsheets became available. They offered an environment where students did not have to learn complicated languages or interfaces, and were not confined to one particular subject or use [BS03].

The first applications of spreadsheets in education were for teaching mathematics, but now spreadsheets are used in all areas of education [BS03]. For example, the Warwick Spreadsheet System covers a wide range of topics in the sciences [Bea92]. Shinners-Kennedy [Shi86] discusses the use of spreadsheets in computer science education and promotes "using the spreadsheet as an operating system" with an aim "to 'build' robust, interactive machines". Assembly programming has been taught using spreadsheets with several advantages over conventional assembly tools because using a spreadsheet gives the potential to observe the internal state and intervene at any instant.

The wide usage of spreadsheets in education indicates that the characteristics of spreadsheets are well-suited to learning. Baker & Sugden [BS03] suggest that it is the middle way between programming (software development) and buying off-the-shelf products (software use)—spreadsheets offer a combined interface for constructing and using an artefact or a model. Steward, in a paper on spreadsheets in mathematics education [Ste94], adds that a spreadsheet exposes the underlying relationships of a model, whereas a program

hides the procedures from the user. Beare states that "spreadsheets promote open-ended investigations, problem-oriented activities, and active learning by students" [Bea92]. It is also true that spreadsheets are popular in education because they are readily available, familiar to existing users, and easy-to-learn for new users [BS03]. Spreadsheets enable us to build on what we already know–exploit our knowledge–in a familiar environment.

This resonates with Rungrattanaubol [Run02] characterising the spreadsheet as an 'open-modelling' environment, as opposed to a closed-world application such as programming. The distinguishing characteristics of 'open modelling' are that it "represents situations, allows meanings to evolve, and offers 'what if' experiment". Simulations and microworlds, although they have been shown to have potential in education, often share the same problems as closed-world applications. As described by Jonassen, "In microworlds, the model is not explicitly demonstrated. Learners have no access to the model and they cannot change it, except to manipulate a set of preselected variables within the model." [Jon06:p14]. Furthermore, research has shown that interacting with microworlds and similar simulation environments does not lead to development and change of mental models [Jon06:p14]. Model construction and manipulation—like in a spreadsheet—is necessary for learners because they learn more than they do from "trying to induce the underlying model in a black-box simulation" [Jon06:p14].

3.4.3 Comparing EM and spreadsheets

The connection between spreadsheets and EM arises in only some aspects of spreadsheet use. Spreadsheets used for business purposes often involve data analysis and visualisation including charts and graphs. In this case the spreadsheet is a tool with a specific goal in mind (i.e. calculating incomings and outgoings for book-keeping). The other use for spreadsheets is more common in education where the spreadsheet can be used for modelling, as described in the previous section. This use of spreadsheets has some of the characteristics of the model-building activity in EM described in Chapter 2. The following example of building a model of the game Sudoku highlights some of the characteristics of model building with spreadsheets that are relevant to EM as an approach to learning.

3.4.4 Illustrating spreadsheets for learning with Sudoku

The game Sudoku originates from Japan, where Sudoku means "one number", referring to the fact that the puzzle grid must contain only one number (1-9) in every column,

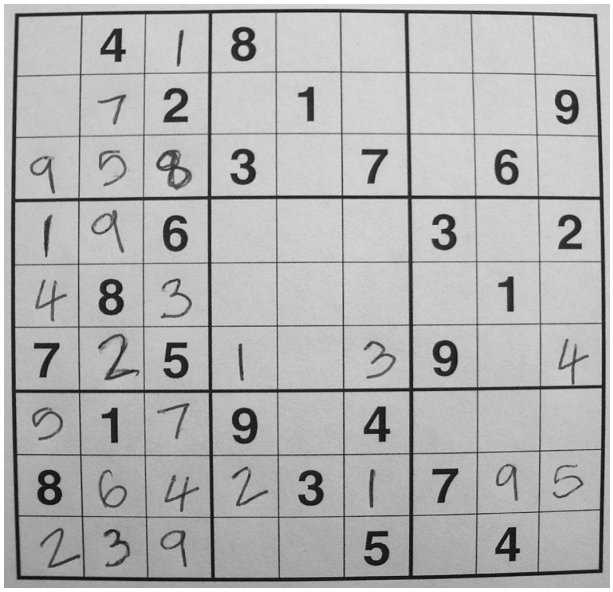

Figure 3.12: A half-completed Sudoku puzzle (from The Guardian [Gua07]).

row and region. Although Sudoku is generally played with the numbers one to nine, the numerals are only symbols and could be replaced by letters, pictures, or any other symbols. Numerous variations of Sudoku exist, but the original game is based on a square grid with 9 columns, 9 rows, and 9 square sub-regions (of 3x3 cells). Sudoku is similar to Latin Squares with an added constraint that each region can only contain one of each number (or symbol). A puzzle starts with some cells supplied as clues from which logical inferences *[obvious or immediate observations]* can be made about the contents of other cells (see Figure 3.12 for an example of a half-completed Sudoku puzzle). The majority of puzzles can be solved by making logical inferences at every step until every cell is filled. The most difficult puzzles contain steps that cannot be logically infered from the current state, and require trial and error to arrive at a solution—these are usually called "impossible" puzzles [Wik07b].

There are a number of Sudoku programs available, offering large volumes of puzzles and help with solving each puzzle. This "help" usually involves offering a "suggested next move" when the player needs it. This then leads to automated solving of the puzzle. For example, Redleg Sudoku (available from `http://www.redleg.biz`) shown in Figure 3.13 offers two supports for solving Sudoku: click the hint button and the program places a number in a square for you; click the solve button and the program instantly fills in the complete solution. This approach might be suitable for fast solving, but it does not give any indication to the human solver how the solution was found. There are several programs, such as Sudoku Puzzle Game and Solver (`http://www.muddyfunksters.com/sudoku/`

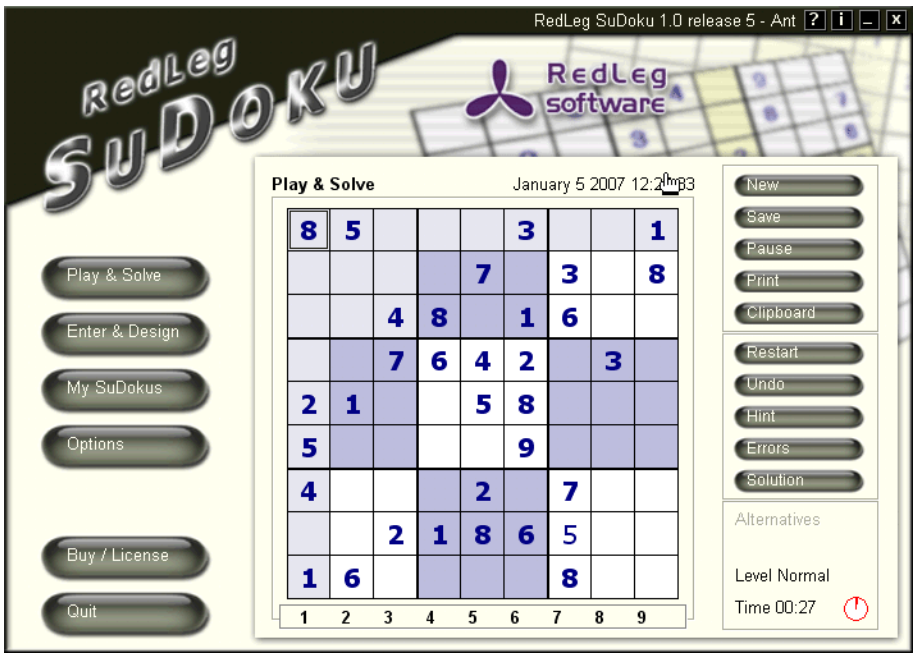

Figure 3.13: A Sudoku program produced by RedLeg Software.

`sudoku.htm`), MPS-Sudoku 2007 (`http://www.ltucshop.com`), and DKM Sudoku online (`http://www.dkmsoftware.com/sudoku/`), that offer similar functionality. Although this type of assistance might be useful in some cases, it rarely leads to significant learning about the nature of solving Sudoku puzzles. It is equivalent to a child who asks how to do something, and a teacher who simply does the thing for the child. (The most likely learning outcome is the knowledge that clicking the "help" button will solve the puzzle!) A better example of a program that supports learning is Sudoku Solver by Logic (available from `http://www.sudokusolver.co.uk/step.html`) which offers a list of logic rules which can be applied in turn to solve the puzzle. This is an improvement because at least at each step the human solver can observe which logic rule was used, and information on a separate page gives an explanation of the more complex rules. However, this program is still automating the solving process such that the human solver plays only an observation role in the process. In terms of offering the human solver support for solving, the simplest program with the least automation is perhaps the most effective. In MPS-Sudoku 2007 each cell offers support for recording the potential numbers for that cell. Clicking on one of the numbers crosses it out, thereby eliminating the possibility that the number can be placed in the cell. It is not automated, the human solver must work out which numbers can be eliminated. It is useful though as it offers some support for solving which is difficult to maintain when solving Sudoku on paper.

3.4.5 Constructing a Sudoku model experimentally

In contrast to the use of a Sudoku computer program, the construction of a Sudoku model adds new dimensions to the learning. In constructing a model there are two aspects to the sense-making outlined in §2.2.3: reinforcing what is already known, and becoming aware of that which was previously unknown. These two aspects are interdependent, and are always found together—when reinforcing what is known, there will be unknowns arising (as discussed in §1.2.3). This should be self-evident when looking at our own learning. Although you would think that we do not always need to reinforce what we already consider to be knowledge, but really, any new learnings must involve that knowledge. Try to use a blank sheet of paper (and no previous knowledge) to learn something new and it will be a struggle. But take a sheet of paper with a Sudoku puzzle and using some knowledge of logical puzzles (even something simple like noughts and crosses) it may be possible to learn something about how to solve Sudoku. The same applies to any type of learning activity, it is usually grounded in something already known—when learning to drive on the road, the learner is generally already aware that cars drive on the left (at least in the UK), that you stop at red lights, that applying the brake helps you slow down or stop, and the learning involves taking this understanding a step further to be able to drive.

Model construction is a process that involves a creator, but the creator is not the only agents to play a part in the construction process. As Bruno Latour suggests, creators have to share their creation with constraints, physical laws, limits of the materials, influences of other creators, needs of users, resources—each exerting opposing pressures on the creation [Lat03]. Creators are not fully in control of the creation. All these factors introduce uncertainty into any creating activity, including model construction. In the words of Latour, "building, creating, constructing, laboring means to learn how to become sensitive to the contrary requirements, to the exigencies, to the pressures of conflicting agencies where none of them is really in command" [Lat03].

An important point is that constructing a model of Sudoku is quite different from the task of writing a computer program to solve (or "help with") Sudoku puzzles. The writing of a program would usually involve a design which may have to be preconceived and thought-out in advance, before implementation is able to commence (as discussed above). The implementation stage would be an exercise in programming, not necessarily in understanding Sudoku. However, constructing a model is about making sense of Sudoku, leaving the potential for exploration of that which is both known and unknown.

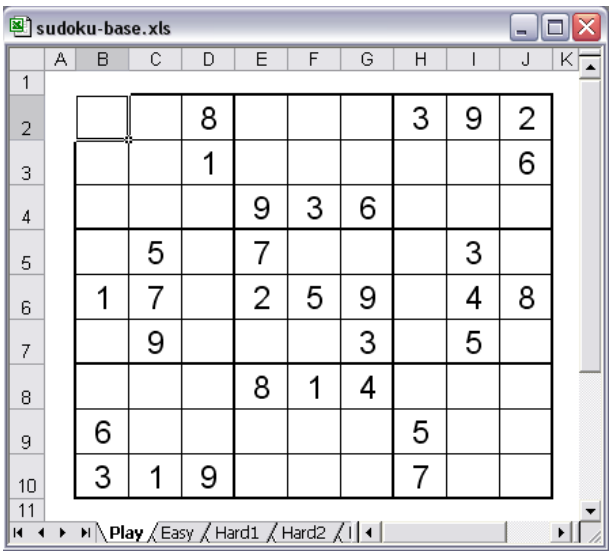

Figure 3.14: An early spreadsheet Sudoku model.

3.4.6 Dependency in spreadsheets for meaningful learning

To construct a Sudoku artefact, a spreadsheet can be created using an environment like Microsoft Excel. There are some obvious benefits to using a spreadsheet as the environment is designed for two dimensional grids of cells. First steps in building a model might involve creating a sheet in which numbers can be placed in a 9x9 grid, and then inserting an initial Sudoku puzzle as shown in Figure 3.14—just as you might do in preparing a sheet of paper for playing Sudoku. In this example, the next step that was taken was to add a cell for each column, row, and region to count the number 1s, 2s, 3s, and so on. This adds some guidance to the solver to determine if a number has already been placed or if a number has been placed more than once (i.e. a mistake has been made). Assuming the puzzle is in the range A1:I9, a dependency for counting the number of 1s in the first column is placed in the cell A11 with the definition `=COUNTIF(A1:A9,=1)`. Any changes to the puzzle grid will also be reflected in any cells dependent on the puzzle grid, so if a mistake is made by inserting too many 1s, it will be immediately visible to the solver.

As discussed in §2.2.2, dependency is an important principle in EM. Firstly, it is important because changes that are made to a model are instantly observable. It allows models to be built without a burden on the modeller to maintain the relationship between elements of the model (the observables). Secondly, dependency definitions tell the model-builder about relationships between the observables. Basically, observables refer to things in the world, and dependencies refer to the relations between the observables. An environment like a spreadsheet (or other EM tools) offers the potential for both observables (i.e. values in cells) and dependencies (i.e. formulae in cells) to be changed, and

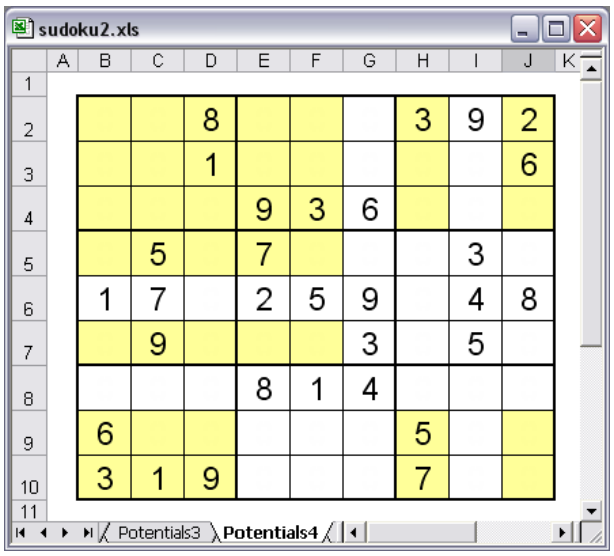

Figure 3.15: Using dependency to highlight the areas where there is potential to insert a '4' in the Sudoku grid.

when changes occur (through acts of agency) dependency makes the effect apparent to the modeller consistently throughout the model. [Geh96]

Returning to the Sudoku model, further dependencies were added to describe other features of the current state of the puzzle. Many of the dependencies are built on top of existing dependencies. For example, for each cell we can determine which numbers can potentially be placed using the column, row and region counts defined above. If, when examining an empty cell, the count of 1s is zero for the column, row and region then the cell may potentially be a 1. If we examine the entire puzzle (of empty cells) for every number (1 to 9), then we have a complete map using dependency of which numbers can be placed in which squares. In Figure 3.15, I have made the background colour of each square dependent on whether a '4' can potentially be placed in the square. The model now becomes more like an artefact to support the solving of Sudoku. Whilst not solving the puzzle, it may give some clues as to where to look to make the next inference in the puzzle. By using dependency we can also deduce why we can or cannot make a particular inference. There is meaning behind the clues, the reason for an inference can be traced from the observables and dependencies relating to the cell—e.g. the definition of the background colour of the cells in Figure 3.15 is dependent on another group of cells determining if there is a '4' in the row, column and area. Compared to the program solvers discussed above which offer the clue without the reasoning, clues in the spreadsheet Sudoku can be investigated, dismantled and manipulated.

By adding more dependencies we might be able to learn more about the nature of

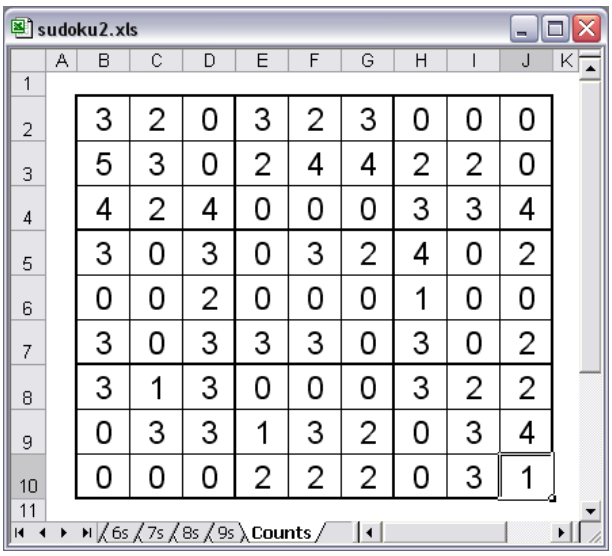

Figure 3.16: Using dependency to make a step in solving the Sudoku puzzle.

solving Sudoku. Using the observables defined above that explain for each cell what potential numbers can be placed, we can create another dependency which counts the number of potential numbers that can be placed. Figure 3.16 shows another layer to the spreadsheet, which counts the potentials from nine sheets like Figure 3.15. With this information, if we find a cell with only one potential number (like the bottom-right square in Figure 3.16), then that cell can only contain one thing and we can make a step by filling this cell with the only possible number (in this case it must be a '4' because Figure 3.16 shows it as a potential, and it must be the only potential because of the bottom-right square in Figure 3.16). Not only is it clear from the dependencies why we can make this step, but it also relates to the way a human may solve the puzzle—i.e. by looking for squares that there is only one possible value to be placed. This is an important feature of EM models, that there is a correlation between the model and the world (or the model-builder's view of the world). The reason this correlation exists is because the model is built on the principles of observables and dependencies that are meaningful connections between the construal and the referent as described in §2.2.2.

3.4.7 The flexible nature of learning using spreadsheets

Another characteristic of the model building in a spreadsheet is the incremental development. As mentioned above, a traditional program is usually prescribed first and implemented second. In model building, the model evolves incrementally as new observables and dependencies are added, or existing ones refined. All this is occurring on-the-fly, in response to changes in our understanding of the domain. We are concerned with the con-

structing activity only in as much as it enables us to make sense of that which is being modelled. The model need never be considered 'complete' or 'finished'. It is an artefact for making sense, for understanding, and for learning.

The incremental nature of modelling leads to a more flexible approach to learning that is characteristic of lifelong learning where it might be appropriate for the learning to occur over a long period of time, revisiting previous ideas and adding to an existing knowledge of some domain. The incompleteness of models and the potential for continuous refinement of models encourages modellers to revisit models, either to explore the domain in more detail (possibly in light of some new experiences) or make use of the model in a new situation. It is possible to combine existing models, reuse an existing model for a new purpose, or take someone else's model to explore from another perspective, as discussed further in Chapter 4. Model-building allows open-ended exploration of a subject that is more suited to learning in an everyday sense.

3.4.8 Spreadsheets as meaningful, flexible and experimental

Spreadsheets are considered easy-to-use and require little training—they appear to be a natural extension of the way we think. Contrast this with programming, which is conceptually challenging for the beginner [Geh96]. Professionals often say that the spreadsheet is too simple to be used for serious applications, but for the application of understanding it is very powerful. The above discussion has highlighted three reasons why this might be so. Firstly, there is the direct interaction and instant feedback that the user experiences using a spreadsheet, which encourages *experimentation*. The second reason is that the spreadsheet can *flexibly* moulded during an open exploration that is likely to involve mistakes and changes of direction. The third reason is that the values and definitions in the spreadsheet correspond to a situation in the world in the eye's of the model-builder—spreadsheets act as *meaningful* mediators. In EM these are termed 'observables' and 'dependencies', and are what we use to construct models—in the same way as a spreadsheet uses values and definitions to construct models such as the Sudoku example discussed above.