Chapter 2

Background

The previous chapter introduced two key areas that are the focus of this work: End-User Development and Empirical Modelling. As the aim is to develop new tools and principles it is important that the existing tools and principles of both areas are understood since they provide the foundation for this work. To that end this chapter will begin by giving a brief summary of key EUD principles, guidelines and relevant existing tools, followed by a similar summary of EM principles and tools.

2.1 End-User Development

Today the focus of the End-User Development community is on End-User Software Engineering (EUSE) [Ko et al., 2011][Burnett, 2009]. It is generally accepted by the community that to a large degree the issue of end-user programming has been adequately dealt with for the time being and now that millions of end-users are programming there are other issues to contend with. These issues relate to the specification, design and testing by end users of their software because one of the major concerns is that of the quality and reliability of end-user developed applications [Burnett, 2009]. However, it seems that this focus is on a minority of end-users who are developing sufficiently large applications in critical situations using EUP technologies. The vast majority of end-

users, especially those at home making personal use of a computer, are still struggling with EUD and hence the focus of this work being to broaden the scope of EUD beyond professionals and education.

As this work is looking at how to merge EUD concepts and Empirical Modelling concepts to produce a plastic environment, the grander issues of design and testing in larger projects are going to be put aside. The focus for this work will remain on the principles and guidelines of EUD and end-user programming, along with the technologies that exist which enable EUD, to see how these can be applied to EM or how EM already uses them.

2.1.1 Principles

There are a collection of key principles that can be found across the EUD literature which are important for enabling end-users to adapt and develop their own systems. The principles give a vague indication of a framework for EUD, although this remains rather loose and poorly formulated. One attempt at developing such a framework is that of *meta-design* by Fischer [Fischer, 2000][Fischer and Giaccardi, 2006][Fischer, 2009]. The most significant and relevant principles are given below.

- Gentle Slope: for an incremental increase in the complexity of an adaptation there should be a similarly incremental increase in the complexity of the mechanisms to achieve it [Dertouzos, 1997][Pane and Myers, 2006][Lieberman et al., 2006]. The main point to gain from this principle is that the user should not suddenly reach a point where they have to modify source code and recompile the application, this is a sudden leap in the complexity involved in modifying the program and is a steep learning curve that many people will be unwilling to overcome. The EUD approach to resolving this usually involves different levels of adaptation, starting with parameterisation for customising certain interface and application features, then allowing users to reorganise components and finally to allow for scripting and modules [Spahn et al., 2008]. Whilst these techniques do produce a more gentle

slope than traditional programming, they are far from a smooth slope and each requires different skills that need to be learned. The more adaptation mechanisms involved the more gentle and smooth the slope of complexity will be, but it is vital that the principles underlying all these mechanisms are related so that there can be an elegant transition which builds upon the previous [Spahn et al., 2008].

- Liveness: Also known as *live editing* [Smith et al., 1995] where changes can be made to a live system without needing to restart the application [Lieberman et al., 2006; Tanimoto, 1990]. This enables users to see and directly experience the consequences of any changes with immediate feedback [Burnett et al., 2001] and allows those changes to be of an incremental nature. It is one of the key characteristics of spreadsheets that has made them so successful [Nardi, 1993, p.88]. In order to achieve live editing a certain degree of robustness is required along with the ability to undo mistakes. It also goes hand-in-hand with the *gentle slope* principle since not allowing for live editing will certainly introduce barriers for the user.

- Directness: by reducing the conceptual distance between the problem domain and software environment the user can more easily identify and resolve their problems by making appropriate changes [Pane and Myers, 2006]. Directness comes from the HCI concept of direct manipulation which attempts to make user interfaces easier to use and is related also to the gulfs of execution and evaluation [Norman, 1998]. It has not been a consideration in the design of most programming languages so the programming world and problem world remain distanced which hinders the problem solving process [Pane and Myers, 2006; Green and Petre, 1996][1]. It is a vital principle in EUD and can take various forms including task-specific languages [Nardi, 1993, p.37] or concrete and directly manipulable objects [Smith et al., 1995].

[1] Programming languages are not usually designed with problem solving in mind.

Table 2.1: EUD Principles

1	Gentle Slope
2	Liveness
3	Directness
4	Evolutionary Development
5	Learning Experience

- Evolutionary Development: sometimes this is described as *design-during-use* or *design for change* [Dittrich et al., 2006][Fischer and Giaccardi, 2006]. Evolution is at the heart of EUD and is the driving factor behind it. Things are always changing, either because the environment is changing or because the focus is changing [Lehman, 1980][Fischer, 2009]. Not only is there change but often it is not possible to adequately identify all requirements to design a piece of software in advance [Swartout and Balzer, 1982], especially in EUD where in the most extreme cases there are no requirements since it becomes an experimental hobbyist activity [Ko et al., 2011]. As a consequence evolution must be supported in software applications and allowing end-users to customise and extend programs is an efficient way to achieve this [Nardi, 1993, p.3].

- Learning Experience: the whole process of EUD can be framed as a learning experience [Repenning and Ioannidou, 2006]. This can be in the sense of learning the tools and how to program, or learning about the specific domain of the application as development takes place. There has been a considerable focus in EUD on educational technologies that allow students to develop simulations or, through domain specific languages, develop their problem solving and programming skills.

2.1.2 Common Approaches

The technologies used for enabling EUD typically fall into one of the five categories which are listed below [Spahn et al., 2008]. It is significant that these approaches are either exceptionally simple but restricted to what the developer anticipated, or involve some form of simplified programming. Despite "gentle-slope" being a key principle it seems there is still a gap between parameterisation and programming. For the most part the simplified programming approaches are also domain-specific and there is yet another gap between them and more sophisticated scripting and programming languages. The bridge between interface customisation and classical programming is rather weak.

- Interface Customisation
- Application Parameterisation
- Programming-by-Demonstration
- Visual Programming
- Natural Language Programming

2.1.3 Existing Environments

Both within the EUD community and outside it there are many examples of adaptable applications making use of the principles and approaches identified above. These applications include GIMP[2], Kate, Firefox, Gmail and many more with most modern applications supporting some level of EUD. Whilst there have been interesting recent developments, such as Service-Oriented Architectures [Erl, 2005] (specifically web services) and the associated end-user developed web mash-ups, for this work of bringing EM into the EUD picture the focus will remain with the major and classic EUD environments and languages. Three of these environments have been identified as key players

[2]GIMP allows for extensive interface customisation, provides many parameters to control the application, allows users to write custom scripts in Scheme and supports user developed modules.

and are of specific interest due to some close connections with Empirical Modelling ideas: Forms/3 [Burnett et al., 2001], Self [Ungar and Smith, 1987] and Subtext [Edwards, 2005]. Other environments which have had some influence on the work include: AgentSheets [Repenning et al., 2000], HANDS [Pane and Myers, 2006], Croquet (now called Open Cobalt), Plan 9 [Pike et al., 1995], Singularity [Hunt and Larus, 2004] and EROS [Miller et al., 2003].

As a research language that is based upon the spreadsheet paradigm, Forms/3 attempts to remove some of the limitations encountered with other similar spreadsheet languages. The spreadsheet paradigm and associated languages are defined by Alan Kay's value rule [Kay, 1984] where a cell's value is defined solely by its formula. According to [Burnett et al., 2001] spreadsheets suffered from two limitations: first is the lack of support for different types and the second is the lack of abstraction mechanisms. It was the goal of Forms/3 to remove these limitations whilst remaining faithful to the value rule and also to the EUD and HCI principles already identified (gentle slope, liveness and directness).

In order to achieve its goals Forms/3 made two basic changes to the standard spreadsheet along with three major additions. The two basic changes are:

1. Removing the grid. Instead of forcing a grid layout onto cells it is possible to position the cells anywhere on the form (aka. worksheet). This enables additional flexibility in visualising results.

2. Giving cells names. Since there is no grid, cells can be given names to identify them.

The three major additions, which provide insight into ways of enhancing Empirical Modelling tools[3], are:

1. Graphical and user-defined types. Cell values may be a graphical type instead of being restricted to numbers and strings as is usually the case in spreadsheets. This

[3]Empirical Modelling tools are also based upon spreadsheet concepts so these enhancements are directly relevant.

allows for graphics without requiring features outside of the spreadsheet paradigm (breaking the *value rule*).

2. Dynamic Grids. The size of the grid may vary automatically and it is possible to give formula to a region without manual copy/paste actions.

3. Temporal streams of cell values. This allows for change over time and hence animation as well as "time travel".

User-defined types are supported in Forms/3 by the use of *type definition forms*[4]. A type is a collection of cells (or cell groups) which can internally reference each other. When a type is instantiated these internal cells may be connected to other cells by a formula reference. Whilst Forms/3 does provide these visual type definition forms and direct manipulation capabilities for creating types, it is all still describable as a textual cell formula and so still follows the *value rule* of a spreadsheet. By sticking with the use of forms, cells and formula for describing types the end-user is not burdened with new concepts such as classes. More recently the Forms/3 type system was extended to include support for inheritance, called *similarity inheritance* [Djang and Burnett, 1998][Djang et al., 2000].

The dynamic grids supported by Forms/3 are similar to traditional matrices and spreadsheet grids, only they are not statically determined. An important feature of these dynamic grids is the ability to specify a formula over a contiguous region (or the whole grid), which removes the formula replication task of the user but also allows for arbitrarily large grids with formulas as they are created in a lazy manner. These dynamic grids offer the same functionality as lists in functional languages since Forms/3 does support recursion. However, when used in combination with the time-varying properties of Forms/3 it has been found that both recursion and iteration are not required and that this improves directness and concreteness which helps with EUD [Burnett et al., 2001].

[4]Also called Visual Abstract Data Type (VADT) forms and are discussed in more depth in [Djang et al., 2000].

Cells in Forms/3 do not just have a single value but consist of a temporal vector which records all the values it has ever had along with the times at which it had those values. With this it is possible for cells to refer not only to current values but to past values and hence enables animation to take place by a cell referring to its own past. Another interesting consequence is that "time travel" is possible where the entire system is reverted to a previous state or where changes to the past can be made with the consequences propagated immediately to give immediate feedback, something which has been called *steering* [Burnett et al., 2001]. It is the liveness of the system which allows for steering whilst keeping the system consistent.

A language which provides further insight into the issue of user-defined types and for providing structure in an end-user friendly way is Self [Ungar and Smith, 1987]. The motivations for developing Self are given in [Smith et al., 1995] where the authors say that:

> *Programmers are human beings... they also need things like confidence, comfort and satisfaction – aspects of experience which are beyond the domain of pure logic.*

One of their key aims was to "integrate the intellectual and experiential sides of programming" [Smith et al., 1995] by providing a consistent and malleable world, not too dissimilar to the notion of a *plastic software environment*, to support exploratory programming [Ungar and Smith, 1987]. As experience and direct manipulation were fundamental ideas the authors decided to devise an environment based upon *prototypes*. Prototype-based object-oriented programming is an alternative approach to class-based object-orientation where new objects are constructed either ex-nihilo or by copying (*cloning*) from an existing concrete instance instead of from an abstract class blueprint [Burke, 2005]. The idea being that this prototype approach "corresponds more closely to the way people seem to acquire knowledge from concrete situations" [Lieberman, 1986], with the focus on the word concrete, something which also appears as

an important concept in the development of Forms/3.[5] It is interesting how the *type definition forms* in Forms/3 seem to be a hybrid concept between class and prototype since these forms are used like a class but can be copied like a prototype [Burnett et al., 2001].

Five key reasons are given in [Ungar and Smith, 1987] for using prototypes instead of classes, these are as follows:

1. Simpler relationships with only a single "inherits from" concept which allows for simpler inheritance hierarchies.

2. Creating by copying where copying is a simpler metaphor than instantiation.

3. Examples of pre-existing modules, allowing a user to examine a typical representative as opposed to attempting to make sense out of a description.

4. Support for one-of-a-kind objects which allows individual objects to be customised directly and independently.

5. Elimination of meta-regress which is a conceptually infinite problem where a class is an instance of a meta-class and so on.

Something considered an important characteristic of Self is its support for live editing, an important EUD principle. According to the authors Self provides "an unusually direct interface to such live changes" [Smith et al., 1995], in part due to the close match and integration between the language and the environments user interface called Morphic [Maloney, 2000]. The directness and liveness of Self depends not only on the purity of the object-oriented language but also on the direct manipulation capabilities of Morphic and its *meta-menu* to make live changes, which creates an "experience of programming that can be learned more easily [by end-users]" [Smith et al., 1995]. The *meta-menu* is available for all graphical objects, called *morphs*, which allows those

[5] "Immediate feedback is facilitated when concrete objects are present in the programming environment [and so] concreteness is a goal as well" [Burnett et al., 2001].

objects to be changed. The meta-menu also provides an *outliner* menu item to show and edit language properties of the object such as slots which include methods and properties.

Not only is Self one of the first and most significant prototype-based languages to focus on the user's experience, but it was also the first interactive pure object-oriented language to have good performance and good responsiveness [Hölzle and Ungar, 1994]. Some of its optimisation strategies have gone on to be used in more well known languages such as Java. What this has done is show that end-user friendly languages do work sufficiently well for real applications.

The final system to be looked at here is Subtext developed by Jonathan Edwards at MIT [Edwards, 2005] and its more recent incarnation Coherence [Edwards, 2009]. Subtext has much in common with both Self and Forms/3, both of which influenced its development. It has spreadsheet like formula characteristics as well as prototype-based cloning as a core concept. Edwards is attempting to "transcend paper-centric programming" and reduce Norman's gulfs of execution and evaluation [Norman, 1998] by making the representation of a program the same as its execution[6]. This involves moving away from just text but at the same time not taking the usual visual programming approaches which have a tendency not to scale well and, according to Edwards, are still *paper-centric*.

With Subtext the programmer is working directly with a tree structure that is both the code and the data, using a suitable graphical interface. Nodes form structures and at the leaves there can either be an empty structure called an "atomic value" or a reference which is much like a spreadsheet formula that returns a node (which may be an "atomic value"). Functions are "structures that react to change" [Edwards, 2005] and use the exact same principle as Core Forms/3 [Djang et al., 2000] where there is a structure with subnodes (cells in Core Forms/3) for the parameters and another subnode for the result with a formula in the result that describes the function. Spreadsheet-like

[6]This similarity of representation in Subtext relates well to the notion of *computation by navigation* introduced in chapter 4 of this thesis.

dependency maintenance is then used to give immediate feedback in response to any change and so Subtext supports the EUD liveness principle.

In some respects, therefore, Subtext is a prototype-based environment that has not embraced the object-oriented paradigm. It is not based on message passing nor does it have imperative methods. Whereas OO merges state and behaviour and Self in particular focuses almost entirely on behaviour, Subtext has achieved the opposite by focusing entirely on state with all behaviour actually being a form of state maintenance. Even function calls are done by copying. In summary then, Subtext is making use of the prototype-based approach to achieve a form of programming that involves direct manipulation rather than indirect textual manipulation. It is perhaps closer to Empirical Modelling than the object-oriented Self language is due to its focus on state.

Edwards has since gone on to develop a new language called Coherence which is based on his concept of coherent reaction [Edwards, 2009] that is much more dependency-like than Subtext. Unfortunately he has recently discovered that his approach is non-deterministic so has abandoned that project (attempting to make it deterministic by moving away from dynamic prototypes to static classes instead) [Edwards, 2010]. Despite this it will be discussed here as an example of a prototype-based and dependency-based language combined.

As with Subtext the Coherence language is based upon a dynamically typed mutable tree and nodes within the tree are created by copying from existing nodes. It uses inheritance mechanisms similar to those found in Self where if a field does not exist then it searches up the inheritance hierarchy to find it. Fields can be defined with a derivation expression which is lazily computed when needed to calculate the value of that field. It is described as being like a formula in a spreadsheet cell and is guaranteed not to produce side-effects. What is unlike a spreadsheet, however, is that derivations are bidirectional so changes to the derived field propagate back to the variables it was derived from. This has been called the reaction by Edwards. It is this that ultimately creates the problem of non-determinism because it is a constraint satisfaction problem,

or, as Edwards states, it is actually a problem of constraint discovery [Edwards, 2009]. Coherent reaction then was an attempt at performing constraint discovery without either reducing the expressive power of the language or involving the programmer. Edwards' higher goal was to deal with the problem of imperative programming where it is up to the programmer to orchestrate the exact order in which all events takes place. It is significant that he chose both the prototype-based and spreadsheet style approaches to achieve this and his justification is given with an example demonstrating the difficulties of execution order in imperative languages. It comes back to ultimately attempting to build a model of coherent state that can deal with change rather than focusing on behaviour.

2.1.4 Guidelines

From their work on AgentSheets, an EUD environment, Repenning and Ioannidou identified thirteen design guidelines for end-user development environments [Repenning and Ioannidou, 2006]. All of these guidelines are, to varying degrees, applicable to a plastic software environment and so will be utilised in the design and development of the new tool. Table 2.2 shows these guidelines.

Of particular relevance is the mentioning of domain-oriented languages and meta-domain orientation. These two concepts appear prominently in Empirical Modelling in the form of definitive notations and an underlying definitive language to bring them all together. Similarly, support for incremental development, decomposable test units and multiple views would be vital for plastic applications and also relates well to Empirical Modelling principles introduced later in this chapter. The fact that these guidelines and the EUD principles fit well with EM is no surprise as both are looking at the same problem but from different angles. Whilst EUD is coming from traditional programming and moving down towards end-users, Empirical Modelling, as will be explained, is coming from the concrete realm of experience and attempting to move up towards programming.

1	Make syntactic errors hard
2	Make syntactic errors impossible
3	Use objects as language elements
4	Make domain-oriented languages for specific EUD
5	Introduce Meta-Domain orientation to deal with general EUD
6	Support incremental development
7	Facilitate decomposable test units
8	Provide multiple views with incremental disclosure
9	Integrate development tool with web services
10	Encourage syntonicity
11	Allow Immersion
12	Scaffold typical designs
13	Build community tools

Table 2.2: EUD Guidelines [Repenning and Ioannidou, 2006]

2.2 Empirical Modelling

The Empirical Modelling (EM) research group is a small group of researchers and students based at the University of Warwick in the UK [EM Website]. The aim of the group is to explore and develop a conceptual framework and associated tools for informal modelling activities that are grounded in experience. Over the years a considerable number of these models have been developed with their tools by various undergraduate and postgraduate students at Warwick [EM Projects]. Their tools have often been used for educational purposes [Harfield, 2008], learning about problems [Care, 2006] and computing [Beynon et al., 2000a]. This section will cover the key concepts of EM, the existing tools and some example models and application areas. The conceptual framework and some of the concepts used in their tools provide an excellent foundation for plastic applications.

2.2.1 What is Empirical Modelling?

The word *empirical* should be well understood as meaning "derived from experience and experiment" and gives a clue to what Empirical Modelling is about. EM takes this notion of experience and experiment right to the core of its conceptual framework to describe a new modelling approach based "firmly on direct, *living* experience rather than formal representations" [Beynon, 2011, p.1]. It is argued that experience is primary and comes before any theory and formal representations can be developed, relating strongly to William James' notion of *radical empiricism* [James, 1912/1996; Beynon, 2007] where to know something is to experience an association between one aspect of our experience and another.

One of the motivations behind the inception of EM comes from trying to develop "theoretical frameworks that do justice to [the] practice" of programming by taking account of first and second factor concerns [Smith, 1987]. Smith's first factor corresponds to the execution of a program and traditional theory, whilst the second factor is about the (human) interpretation of programs. It is the gulf between the formal and informal

worlds. Beynon and Russ discuss these two factors in [Beynon and Russ, 1992] where he says that "an adequate theory of computation must allow a high degree of interaction between these two factors". It is noted by Beynon and Smith that computer science only talks about the first factor and that "curiously" the semantics of a program do not consider the second factor, that of interpretation. The consequences of this lack of concern for the second factor are eloquently put by Black:

> *"the drastic simplifications demanded of success of the mathematical analysis entail serious risk of confusing accuracy of the mathematics with strength of empirical verification in the original field"* [Black, 1962] as cited in [Beynon, 2011]

Focusing only on the abstract mathematical world and ignoring the concrete reality we inhabit will likely mean that any model, program or theory developed in such a way will be suspect when it comes to being interpreted. In practice the concrete is not ignored:

> *"In devising a mathematical model of a tree, the mathematician adopts a constrained way of observing features relevant to a functional objective but may first need to identify suitable features and patterns of behaviour derived from exploratory experiment"* [Beynon, 2011, p.5]

If the word "mathematical" is replaced with "computer" and "mathematician" with "programmer" then the above statement by Beynon accurately describes the need for a software development process with requirements (a functional objective) that need to first be formulated. Exploratory experimentation is a term used by Steinle for experimentation that takes place before any well formulated theory exists, in contrast to experimentation that is "theory-driven" [Steinle, 1997].

> *"Exploratory experimentation, in contrast [to theory-driven experimentation], is driven by the elementary desire to obtain empirical regularities and*

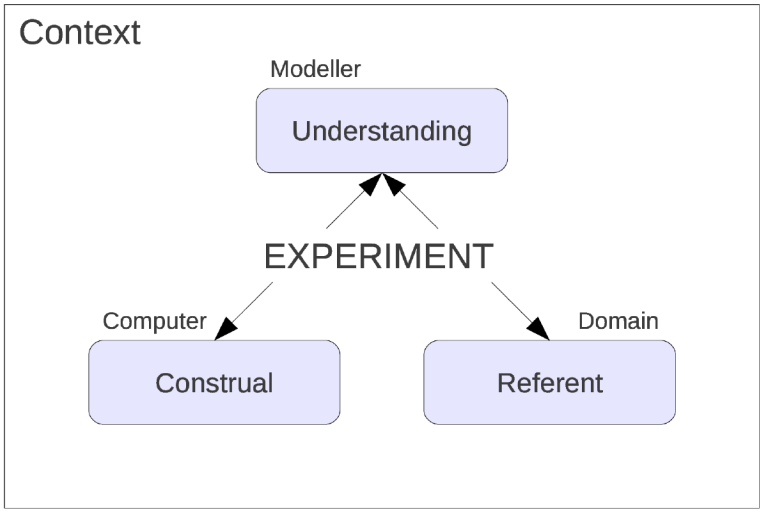

Figure 2.1: Original EM Conceptual Diagram

to find out proper concepts and classifications by means of which those regularities can be formulated." [Steinle, 1997]

Exploratory experiments can be found throughout the scientific community, with a well documented and interesting example being the experiments done by Faraday in trying to develop a theory of electromagnetism. Gooding provides an account of Faraday's work which attempts to discover how Faraday was able to gain important understanding through the use of physical, metaphorical models of otherwise invisible forces [Gooding, 1990]. Faraday uses exploratory experimentation to eventually develop a theory. It is clear from this, as well as from Black's comment, that everything is, and should be first and foremost, grounded in experience. This is certainly the tenet of Empirical Modelling which has borrowed ideas such as that of *construal*[7] from Gooding's work on Faraday.

So Empirical Modelling is attempting to develop a framework for pre-theory exploratory modelling which enables the modeller to identify "empirical regularities" and gain sufficient understanding to construct a formal account. Experiment is to be used to

[7] A construal is a provisional, personal sense-making entity [Beynon, 2011, p.1] and originated in the work of David Gooding [Gooding, 1990] regarding the modelling activity of Faraday.

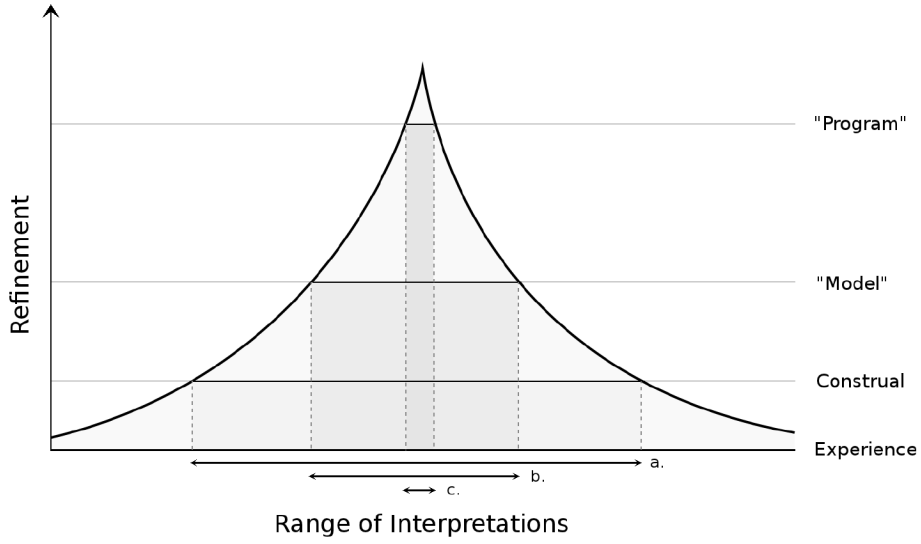

Figure 2.2: Refinement from experience to "program"

bridge the gap between immediately experienced and circumscribed behaviour [Beynon, 1994]. Figure 2.1 shows how this has been conceptualised as a process involving four components which co-evolve in a live fashion to construct and refine models whilst allowing the modeller to gain understanding [Beynon, 2011].

The diagram in figure 2.2 is intended to illustrate how, in EM, an artefact is refined from being initially an open-ended experience to being a "program" with restricted interpretations and *ritualised interactions*[8]. The curve shows, in a much simplified way, the boundaries of what the modeller considers as meaningful interactions and interpretations at a particular stage of refinement. As the artefact becomes increasingly refined (moving up the y-axis) through a process of exploratory experimentation the range of possible interactions and interpretations becomes increasingly restricted, shown by a,b and c. Of course, the process is considerably richer than indicated in figure 2.2.

> *"the model migrates... from provisional to assured, subjective to objective, specific to generic, personal to public. To support this migration, the model has to be fashioned, via interactions both initiated and automated, so that*

[8]See Harfield's PhD thesis for more on ritualised interactions [Harfield, 2008, p.33].

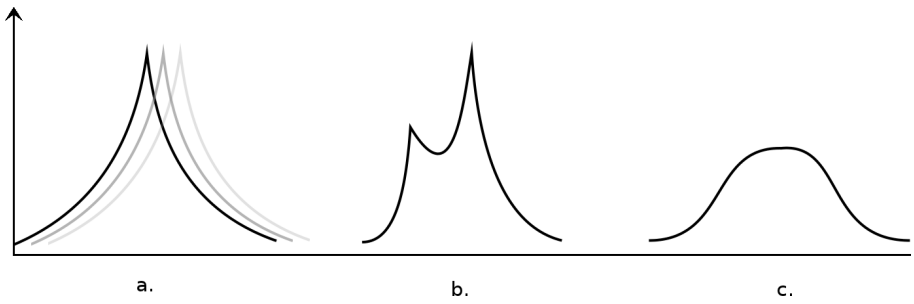

Figure 2.3: Problem Shapes

> *the views of many agents - human and non-human - are taken into consideration. This process of construction is in general open-ended and never absolutely resolved."* [Beynon, 2011, p.17]

An analogy of devising a walk has been used in [Pope and Beynon, 2010b] to explain the evolution of an EM artefact. Initially any walk plan is entirely unformed and the possibilities are almost unrestricted, which corresponds to being at the experience level in figure 2.2. Gradually the walk planner will decide on a region and then maybe a specific mountain to walk on. This is a refinement which moves up the diagram to perhaps be considered a *construal* or "model". The process continues until the planner has a precise set of instructions for paths to follow and this would be at the level of refinement expected of the "program" level. What this analogy shows is how initially any plan is provisional, subjective and personal but will be refined by an exploratory process until it is assured, objective and public. The walk plan is initially at the mercy of the planner's imagination (within certain limits) but eventually becomes something which can be described to others and followed in an objective fashion.

Figure 2.3 shows how in practice the curve showing the boundary of meaningful interactions and interpretations is more complex than that of figure 2.2. a) shows how this curve, which corresponds to the environment, context and modeller's goal, can change with time. This would mean that a well refined artefact may, without itself changing, become incorrect with respect to some criteria. b) is a situation where

it is possible to refine an artefact to a point and get "stuck", requiring a degree of backtracking to a previous and less refined state in order to explore and experiment a different way. Finally, c) shows a scenario where refining to an assured, objective and public artefact may not be desirable. What is important then is for an ability to move smoothly from construal to model to program and back again. The categories of construal, model and program are blurred concepts attempting to name specific stages of a continuous process.

An important note to make is that the terms "program" and "model" as used in figure 2.2 are not identical to the traditional concept of program and model. A traditional program will typically have fixed functionality to enable optimisation, whereas the EM "program" still has a flexible functionality (possibly at the expense of not being optimised). Unfortunately perhaps, computer science and the software industry have widely assumed that problems must be abstracted before they can be solved, with increasing layers of abstraction being the solution to complexity. Computer Science has stuck with the first factor, ignoring the second and so results in inflexible programs.

> *"Whereas conventional modelling uses abstraction to simplify complex phenomena, EM generates an interactive environment in which to explore the full range of rich meanings and possible interpretations that surround a specific ... phenomenon"* [Beynon, 2011, p.3]

One of the major challenges, besides developing an appropriate conceptual framework, is finding ways of supporting the migration described above. A more in depth discussion of the philosophical and conceptual aspects of EM can be found in [King, 2004; Beynon, 2007; Beynon and Russ, 1992; Beynon, 2011]. These conceptual issues are vital for providing a solid foundation so are discussed in chapter 6. However, the focus of this thesis is to explore ways of achieving the migration from construal to program, something not currently practical with existing EM tools, as will become clear, but which is believed to be possible with the EM conceptual framework in general.

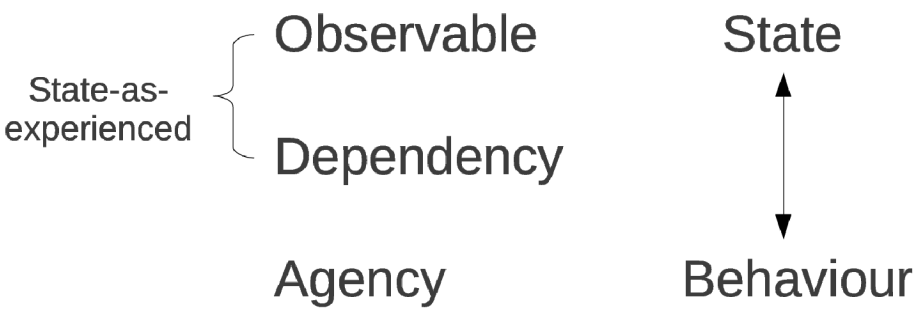

Figure 2.4: EM ODA Framework

2.2.2 The Principles

One of the key objectives of EM is to provide a solid conceptual framework and philosophical account of this kind of exploratory, experiential modelling activity on a computer. An experiment involves the observation of and interaction with state and so Empirical Modelling principles are "fundamentally concerned with modelling state" [Harfield, 2008, p.31]. Since the state being modelled is "empirically apprehended" and is somewhat different to traditional program state, it has been called *state-as-experienced* [King, 2004, p.30]. In addition to modelling state the notion of agency, the observation and interaction, needs to also be considered and so becomes the second central concept.

The construction of artefacts that relate to *state-as-experienced* in the referent is the process being described by figure 2.1. In the diagram there is only a single modeller involved, but there is little reason not to consider multiple human agents being involved and work to that effect can be found in [Sun, 1999; Chan, 2009; Beynon and Chan, 2006].

The concept of *state-as-experienced* can be split into two: observables and dependencies. The motivation for this split lies in the fact that we not only observe quantities or qualities of experience but also the associated relationships between them. Therefore there are three core concepts in Empirical Modelling which are well expressed in [King, 2004]:

- Observable: "something which has a value or status to which an identity can be

ascribed". There are no restrictions on what this might be and certainly does not need to be numerical. It is not a mathematical variable due to its concrete nature, but may have some connection to variables in traditional programming languages.

- Dependency: "a relationship between observables such that interaction with one observable leads indivisibly in our experience to a change in the other". These relationships are what allow changes to propagate and crucially are what enable experimentation to occur. They are the "experientially-mediated" associations which, from radical empiricism [James, 1912/1996], provide meaning.

- Agency: "an agent is projected on to the referent as something that can change the state of the model in some way by manipulating observables and the relationships between them". Each agent may have a different view and interpretation of the model of state. Agents may be human or non-human, there may be one or many and they may also act concurrently [Beynon, 1997a].

The first two concepts enable a rich model of state to be developed which remains coherent in the presence of change and also, due to the way it is developed, remains meaningful to the modeller. A vital component not often highlighted in the practical aspects of EM is the significance of an "observational context" [Beynon, 1994]. What observables and relationships are important to a given agent at a given point in time will depend on a potentially shifting context of observation. An example given in [Beynon, 1994] is of a Newtonian model for projectile motion. In this model certain observations such as air-resistance are ignored initially but at some future date the context may change and such observations may then become important. A classic example given by Beynon in lectures on Empirical Modelling is a line drawing model of a filing cabinet (figure 2.5) which may also be interpreted as an LCD digit [Beynon, 1990]. Which interpretation is to be taken will influence what observables and relationships exist and what agent actions may be meaningful (it makes sense to open a filing cabinet but not to open an LCD digit). At any time the context could shift from filing cabinet to LCD.

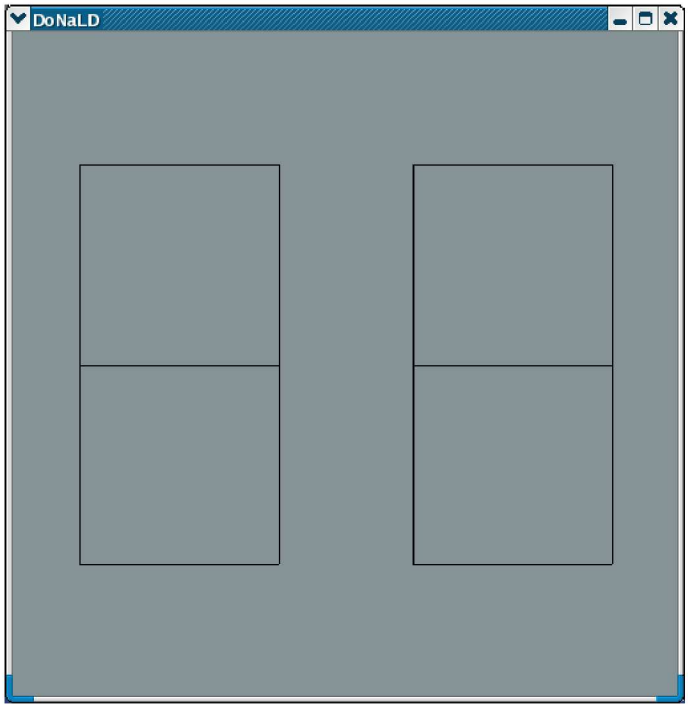

Figure 2.5: Filing Cabinet and LCD Digit

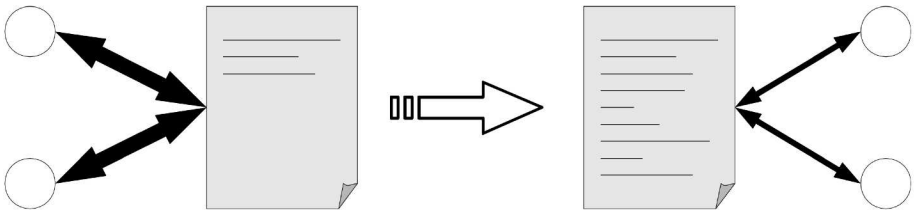

Figure 2.6: Agents interacting with a definitive script. As the script matures agent interactions are restricted.

State-as-experienced (observables and dependencies) is embodied in a *definitive script*, a concept that has been with Empirical Modelling since Yung developed EDEN [Yung, 1990] and which has been explored in depth in [Rungrattanaubol, 2002]. All interaction with the model is conceived as making redefinitions in the script. A definitive script may consist of a collection of scripts, each of which may use a different *definitive notation*[9] that is task-specific. Figure 2.6 shows how agents interact with scripts and that over time these interactions become *ritualised* and restricted.

2.2.3 Current Tools

Whilst EM is primarily a conceptual framework there has been considerable tool development to support such a modelling activity on a computer. The only Empirical Modelling tool currently in use is Tkeden, originally called EDEN[10] by Edward Yung who first developed it [Yung, 1990]. EDEN was intended to be a "general-purpose language supporting definitions" [Yung, 1990, p.6] which borrowed much from spreadsheets and C. It is useful to note that EDEN was developed around the same time as both Self and Forms/3 discussed previously. Since its inception, EDEN has been extended by many developers and used by hundreds of students and academics [EM Projects]. Ashley Ward gives a detailed up-to-date picture of the EDEN tool in chapter 4 of [Ward, 2004].

Additional tools have been explored, including the Abstract Definitive Machine (ADM) by Mike Slade [Slade, 1990], Definitive Assembly Maintainer (DAM) by Richard Cartwright [Cartwright, 1999] (along with a Java version called JaM and subsequently JaM2) and finally the Definitive Object State Transition Engine (DOSTE) [Pope, 2007] which was developed by myself as Empirical Modelling coursework. The original DOSTE is not the same as the work given in this thesis but was the origin of some of the concepts. Again, these tools (except DOSTE) have been reviewed in detail by Ward in [Ward, 2004].

[9] A *definitive notation* called ARCA was the origin of Empirical Modelling [Beynon, 1985] and so this concept has been involved since the beginning. It is intended to be a specific algebra for a particular set of problems, in this case Cayley Diagrams.

[10] Engine for DEfinitive Notations.

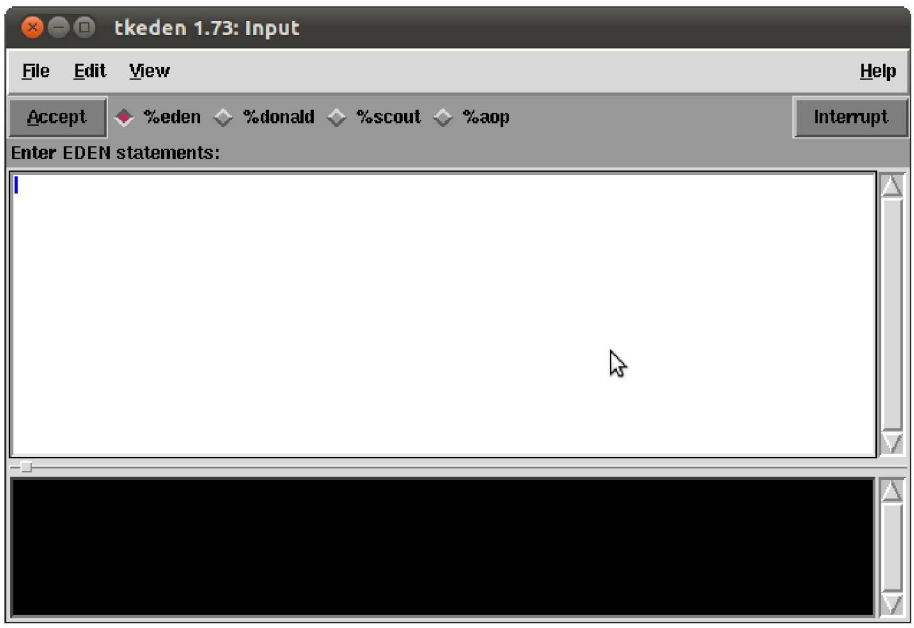

Figure 2.7: Tkeden Input Window

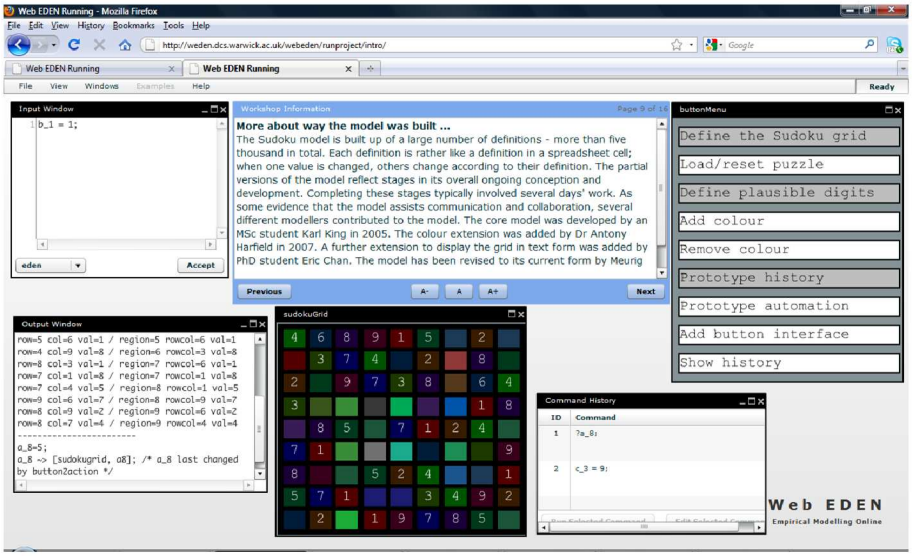

Figure 2.8: Web interface to Tkeden developed by Richard Myers

For the purposes of brevity only the EDEN tool will be discussed here, although the ADM with its object-like concepts has been influential. Eden, the definitive language of the EDEN tool, uses a less strict C-like syntax with the addition of appropriate syntax for describing dependency definitions and dealing with agent actions. A version of EDEN called *tkeden* provides a simple interface as shown in figure 2.7. More recently a web interface to *tkeden* has been developed by Myers [Harfield et al., 2009] and is shown in figure 2.8. The key difference between the various EM tools is how they interpret and implement the three core concepts. Each tool takes a different approach, especially true for agency. In EDEN the ODA framework has been implemented as follows:

- Observables: The interpretation given to observables in EDEN is that of a flat space of variables with a specific and limited set of available types along with a C-syntax identifier. The typing is not rigid as no observable gets declared to be a single type but can change type at any time. The available types are: integers, reals, strings and lists. There has been some work to extend the original EDEN so that observables can be grouped into something called a *virtual agent*. This virtual agent feature has not proved successful or reliable and so remains unused in the most recent models.

- Dependencies: Expressed as functional like formulas using a C-syntax and associated with a specific observable in a way that is similar to a cell being given a formula in a spreadsheet. All references to observables in the definition are considered to be a dependency and so when any of those set of observables changes this particular formula/definition is marked as out-of-date and re-evaluated when next accessed. All of this dependency maintenance occurs indivisibly to the modeller and any agents.

- Agency: There are two distinct forms of agency to consider in EDEN. The first is the modeller (a.k.a. the human user). The modeller can interact with a model in a variety of ways including a textual input window (cf. figure 2.7) or with mouse

clicks. In the textual input window they may enter script in one of several different notations depending on the domain of interaction or observation. Such notations include the basic Eden notation and notations for line drawing (DoNaLD), window management (SCOUT), 3D graphics (Sasami) and relational databases (EDDI) as well as a newer notation for constructing custom notations (AOP) [Ward, 2004]. In addition to the modeller there is some scope for automated agency. Automation is achieved through the use of triggered procedures which can request to be called whenever a specific set of observables has changed (either directly or via some dependency maintenance). Such triggered procedures can then use conditional statements, loops and other similar constructs to observe and make changes to the state of the system by changing values of observables or giving them new definitions.

Something that needs further elaboration is the use of multiple notations. The purpose of this is to provide different algebras, in the form of data structures and operations, that are specific to a particular domain such as line drawing. All the notations convert down to the underlying Eden language to build up the required sets of observables, dependencies and agents which correspond to the more abstract concepts in these "higher" notations.

2.2.4 An Example Model

To illustrate the Empirical Modelling process and show the concepts in action, the digital watch model will be briefly discussed. This model was originally developed in *tkeden* by Beynon in 1992 based upon a state chart by David Harel in [Harel, 1988] of a digital wrist watch. Over a period of eight years four different people, including Beynon, were involved with this model, extending, refining and revising it (cf. figure 2.9) [Fischer and Beynon, 2001; Roe et al., 2001; Beynon and Cartwright, 1995; Roe, 2003]. Initially the model involved only the state chart and a digital display, developed over a period

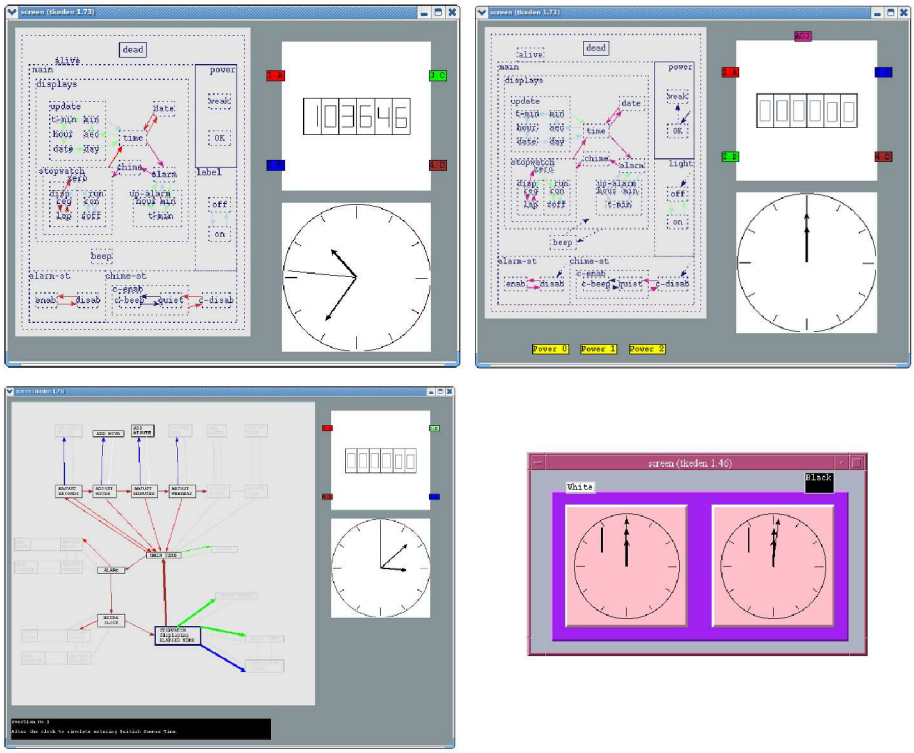

Figure 2.9: Evolution of the Digital Watch. Top-left: Cartwright 1995, Top-right: Fischer 1999, Bottom-left: Roe 2001 and Bottom-right: Cartwright's Chess Clock 1995

of about 3 weeks using a continually running program on an office workstation.[11] The model was developed incrementally by adding in small groups of definitions and agents to add some new visual component or agent behaviour. Interestingly Beynon states that "it was sometimes useful to develop portions of the script independently, or to trace problems by extracting pieces of the script and exercising them in isolation" [Beynon and Cartwright, 1995], which relates well to some of the EUD guidelines and indicates the flexibility of the tools. Beynon's resulting model consisted of 2000 lines of script with around 1000 definitions and was then used to help teach EM to MSc students at the time.

Following this, students independently and without much help from Beynon, were able to experiment with the model to learn how it worked and subsequently extend it in various ways. Some of these alterations were examples of a radical shift in *observational context* that involved adapting the model to an entirely new purpose, such as Cartwright's Chess Clock shown in figure 2.9 [Cartwright, 1995]. Other changes involved the addition of new buttons and functionality to, for example, allow the date and time to be changed.

Changing the model is a relatively simple activity. The modeller can enter new definitions, written in one notation or another, into the *tkeden* input window and click "accept" to see the consequences of that change immediately. Such a change might be to make the hour hand of the analogue display depend upon the minute hand so that if the minute hand is changed the hour hand also changes by dependency [Fischer and Beynon, 2001]:

$$\text{angle\_hour\_hand} = (\text{angle\_min\_hand}/2\pi) * (\pi/6)$$

Other changes may be made to the visual representation, such as the colour of a button. Not only do these changes incrementally develop the model, they also allow for experimentation and testing. Normally, in the more developed versions of the model,

[11] Except for the occasional crash, "but several days typically elapsed between such events" [Beynon and Cartwright, 1995].

agents are responsible to operating the various watch mechanisms. However, this can be interfered with by the user to, for example, simulate a malfunction of the alarm or battery. Such interference is outside of the preconceived pattern of interaction [Fischer and Beynon, 2001]. This kind of experimentation serves several purposes: to gain understanding of the model, to learn something about the referent or to check how well the model matches with the referent (does a particular interaction cause a similar effect in the model as it would in a real watch). Unlike some EM models the digital watch model is based upon a well understood device and so is not a truly creative and exploratory endeavour unless it was to look at new kinds of watch design or other changes of context.

2.2.5 EM and Software Development

There has been considerable work on using Empirical Modelling as a new way of developing software, with EDEN being partly developed under the title of "a Paradigm for Exploratory Programming" [Yung, 1993] and Ness who looked at "Creative Software Development" [Ness, 1997], along with other work by Beynon et al. [Pope and Beynon, 2010b; Beynon et al., 2008; Beynon, 2011]. A common response of those from industry (and from students) who are introduced to EM is that it provides a means of requirements elicitation and perhaps prototyping but nothing more. By taking this view they are missing the point and the potential of EM.

Traditionally developed software relies on "pre-existing ... theory or established empirical knowledge" [Beynon, 2011, p.1] as "formal representation draws on previous experience" [Beynon, 2011, p.18] which may then be appropriately abstracted. It is only possible to generate abstractions once sufficient understanding has been obtained. In contrast, EM looks to support not only the traditional software (by in principle supporting abstraction) but also software where there is no pre-existing theory or pre-computer precedent by allowing for experimental exploration that is yet to be abstracted. In effect it is allowing computer science to be an experimental science [Milner, 1986].

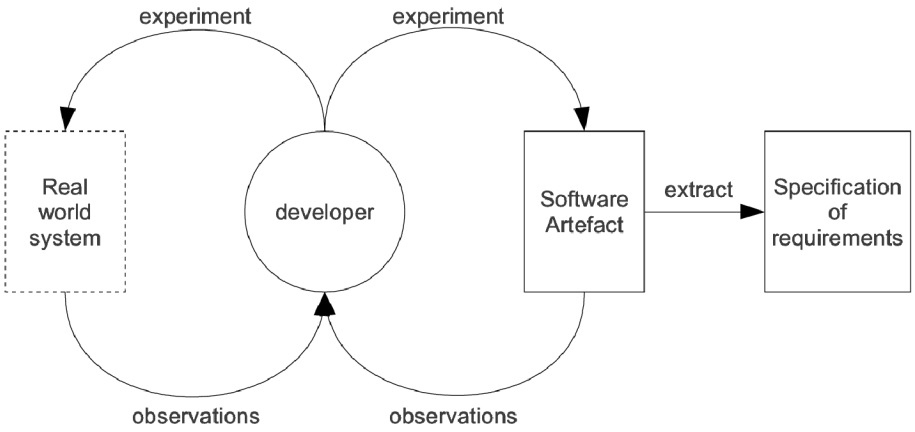

Figure 2.10: EM Software Development Process, based upon diagram in [Beynon and Russ, 1995]

So EM does begin before any traditional requirements stage and does involve a form of prototyping, however, there is little reason that, with appropriate tools and framework, these models cannot evolve into applications without a big leap from requirements to design to implementation. The diagram in figure 2.10 shows the EM development process as conceived in 1995. What is significant is that a "specification of requirements" is extracted from a model to then generate a fixed functionality program, but nothing beyond that has been elaborated. The process at the time tried to integrate the requirement, specification and design phases of a development, but not the final implementation which was to be done by some unknown transformation from model to program [Beynon and Russ, 1995].

There has been little progress on this since 1995 with most work becoming centred on using *tkeden* or mapping the conceptual framework onto specific domains such as product design [Ness, 1997] or educational tools [Harfield, 2008]. There have been attempts made (but not well documented) on an automatic translation process from model to program. Some work was done by Sun on distributed modelling and software development which involved adapting *tkeden* to produce a distributed version called *dtkeden* [Sun, 1999]. Recent work on EM and constructionism also gives some in-

sight [Beynon and Harfield, 2010]. The most successful models in software development terms are the temposcope [Beynon et al., 2000b] which was briefly used as a finished product, and colour sudoku [Beynon and Harfield, 2010] that has had widespread appeal. Full transition from model to program has been a goal for some time but as yet not achieved. Scaling up is perhaps the limiting factor.

2.3 Miscellaneous Technologies

Despite the existence of structured data models, such as relational databases, there is a need to represent data that cannot be constrained by such strict pre-designed structures. The World-Wide-Web is perhaps one of the key motivations for developing the notion of *semi-structured* data (also called *self-describing* data) as it is difficult to see how this could be constrained by a schema [Buneman, 1997]. Another issue apparent from the beginning is the inability of users to browse a traditional database without writing a query which requires knowledge of the schema [Buneman, 1997], making exploration of the data difficult. The approach taken by researchers and industry for dealing with semi-structured data is to represent the data as an edge labeled graph. Today the most well known and exceptionally popular approach is that of XML. The notion of semi-structure will become relevant when considering ways of improving EM tools (cf. §3.4.1).

Most often based upon XML, dependency injection is a way of configuring components and attributes of a program by using an external description which links the components together [Chiba and Ishikawa, 2005]. For example, XML can be used with Java to connect various objects together at load-time rather than embedding those connections in the source code. This enables the reconfiguration of components without recompilation of the program and is an attempt by developers to reduce the dependencies hard-coded into the application, enabling component reuse. These dependencies are then available for modification by the developer, although not usually in a live fashion.

The use of dependency as found in EDEN can also be seen in commercial products other than spreadsheets. Microsoft's Windows Presentation Foundation (WPF)

includes the concept of *dependency properties* [Cox, 2008; Harfield, 2009]. Dependency properties enable simple connections between properties to be made but it is difficult to describe more complex relationships. Additionally, WPF dependency properties are not interactive but get compiled into the application, there is no possibility of changing the dependencies live. Flex is another technology that is also employing the use of dependency in describing interfaces.