Empirical Modelling as an unconventional approach to software development

Nicolas Pope
University of Warwick
n.w.pope@warwick.ac.uk

Meurig Beynon
University of Warwick
wmb@dcs.warwick.ac.uk

Abstract

Flexible modelling tools are needed to address the demands of modern software development. Historically, software development was concerned with the classical computer as the sole computational agent. Here the environment and problem are both well understood and so the emphasis is on abstract specification. There is, however, an increasing awareness of the central importance of sense-making activities throughout the software lifecycle. Empirical Modelling (EM) provides a broad conceptual framework within which to approach software development of this kind. We briefly discuss our progress towards developing tools that enable this initial sense-making activity and discuss how they could be utilised for software development. We also outline our vision of how software development could be radically altered by considering EM principles.

1. Introduction

The term "modelling" has exceptionally rich connotations. In relation to software development, it can refer to activities in all aspects of the traditional life cycle. This is illustrated in UML, where there are ingredients that relate specifically to modelling processes to be implemented by programs, and others to the structure and configuration of the computational agents to be programmed. Different emphases in modelling activity may be appropriate according to the application: whether we are concerned with 1-person programming, or reactive systems [9], whether our system development is an exercise in routine or radical design [10], whether the solution is designed by an individual or collaboratively.

Flexible modelling tools are needed to address the diverse varieties of modelling activity that modern software development entails. To put the research outlined in this position paper in perspective, it is helpful to distinguish different varieties of modelling that are represented in software development in general. In broad terms, this modelling activity can be classified according to different states of knowledge about the domain in which the software system is being developed. In framing this classification, the term 'computational agent' is used to refer to any component of the system whose behaviour is specified by rules conceived by the developer. Such an agent may be a conventional computing device, a human agent observing the conventions set out in a 'user manual', or a programmable peripheral capable of sensing and acting on its environment.

- Type A: The nature of the computational agents within the application domain may be very clearly identified. In this case, the emphasis is on specifying exactly what protocol the computational agents follow.
- Type B: The computational agents and the core features of the application domain are clearly identified, but the extent to which situations arising within the domain are constrained and the capabilities of the agents and the interfaces supporting their interaction are still to be fully explored.
- Type C: The potential for agency that can support computational interpretation within the domain is unexplored, and the framework within which the system is to be conceived has yet to be identified.

Type A modelling is directly related to 'specification' and what is termed 'programming' in the narrow sense. High-level programs can be seen as models of this type. Software development was initially concerned with the classical computer as the sole computational agent. In that context, the primary emphasis was on the problems of accurately specifying procedures in the abstract, since the context for interacting with the computer and interpreting this interaction was so tightly constrained.

Type B modelling is motivated by the realisation that programming of computational agents (however well understood in the abstract) is only effective in practical situations if a whole range of concrete engineering factors is taken into account. Such factors include assumptions about the application domain, the speed and response of devices, the characteristics of the interfaces and the perceptions, knowledge and skill of the users etc. Type B modelling is associated with a 'software engineering' rather than a 'theory of computation' perspective on progamming.

Type C modelling reflects awareness of the limitations of routine design in respect of modern software system development, and the problems encountered in modifying even well-engineered software systems to meet new functional requirements. It is motivated by many factors: new technologies, new paradigms for interaction, new applications outside the scope of traditional business, science and engineering, and the speed of change. Type C modelling is associated with the preliminary sense-making activities that precede the articulation of requirements.

In broad terms, the practice of software development has evolved from type A towards type C modelling. Conceptually, in the context of modern software development, and the call for 'flexible modelling tools', it is more appropriate to view these activities as moving in the opposite direction, from type C to type A. Once a system is sufficiently complex, all three types of modelling are entailed in its development and maintenance. Type A modelling is

FlexiTools at SPLASH 2010 18th October, Reno, Nevada, USA

closest to the computer, and to the formal abstract perspective on computation. Type B modelling invokes experiential and physical ingredients that demand empirical investigation. Type C modelling is the furthest from formalisation, and on the face of it is not well-suited to the application of computers.

Our perspective on software development is informed by a well-established programme of research on Empirical Modelling (EM), as developed by the second author and his collaborators [1]. Where the potential application of EM to software development is concerned, a crucial contribution has been made by the first author through his development of the DOSTE environment. DOSTE supplies the basis for a radically new kind of software development that can be seen as extending the range of EM, where the principal focus to date has been on modelling *environments* and *agents*, to encompass modelling *processes* and *objects*. EM provides a broad conceptual framework within which to enact software development through transition from type C to type A modelling. As we shall briefly outline in this paper, environment-agent EM gives a grounding and proof-of-concept for a study of software development using process-object EM that has yet to be fully explored.

2. EM for sw development: orientation

The aspiration in EM is to build artefacts which exhibit interactive characteristics similar to those observed in the situation to which they refer. The relation between an artefact and its referent is established through a close correspondence between dependencies, observables and instances of agent action. Specifically, different kinds of agent interaction with the referent have counterparts in the model that are recognisably congruent in that they disclose similar dependencies between observables. The full elaboration of this notion is beyond the scope of this paper - it is central to the body of publications at the EM website [1]. A crucial aspect of the approach is the emphasis that is placed upon the experiential nature of the correspondence between an artefact and its referent. This is a radical departure from the conventional functional and operational manner in which a computer program is interpreted. It means that the interpretation of an artefact is open and fluid. For instance, it is subject to evolve over time ("facility in recognising dependencies can be learned"), can be dependent on the observer ("relationships can only be discerned if the observer isn't colour-blind"), and on the specific situation within which interaction and observation is being conducted ("whether changes to observables can be identified may depend on the level of lighting").

Exploiting EM makes it possible to rethink software development as a variety of type C modelling. A key contribution of EM has been to give philosophical grounding to informal sense-making that can justify the claim to meaningfulness without abstraction and formal knowledge representation [4]. This draws on the *radical empiricism* of William James [11], making a direct appeal to the idea that all knowledge is rooted in connections between one experience and another that are themselves given in experience. From an EM perspective, software development can be seen as analogous to devising a walk across terrain that is unexplored or little documented. The finished program corresponds to the route plan and instructions for following the walk and actually running the program is like following these instructions. The pivotal point of our analogy is that devising such a walk is rooted in an exploratory activity that is directly enacted within the external environment. We experience the lie of the land and its distinctive concrete features, but have yet to abstract the conception of the walk, to determine what direction our walk will take, and which landmarks it will follow.

A precedent for software development of this nature can be found in the use of spreadsheet models to create business applications (as discussed by Nardi [12], for instance). Such models make a direct connection between the current state of a computer model and the current situation in the application domain. This connection with real-world semantics is so immediate that to a large extent it can be experienced by all those who interact with the application, whether as naive users, sophisticated users, managers, developers etc. The way in which the software application is then created is closely linked to the business practices that surround it, each helping to shape the other. A common feature of such software development is that activities and processes that are first developed informally through using spreadsheets to model the business environment evolve seamlessly into program-like activities that are supported by user protocols and appropriate macros.

Spreadsheets are limited as tools for software development. The metaphors they afford for modelling situations are well-suited to environments such as are encountered in science and business where there are well-established relationships between numerical data values. They are not so well-adapted to deal with the data structures and interface components that feature in general software development, nor with other kinds of relationship that feature in everyday experience. Whilst additional modes of data visualisation can broaden the expressive range, these techniques are not enough to enable experiential connections between the model and its referent that are needed to sustain type C modelling in a more general setting.

3. Environment-agent EM for sw development

EM is a general conceptual framework for studying how computing technology can support type C modelling. In EM, the primary activity is making *construals* - interactive artefacts that serve a role in sense-making (cf. [8]) - rather than programs. Applying EM to software development involves making construals that metaphorically represent the environments in which computational agents interact as construed by the modeller. The construction of the EM construal is grounded (in much the same way that the current state of a spreadsheet model is grounded) through direct appeal to the correspondence between interactions with the construal and interactions with the external environment it purports to represent. Reproducible trajectories can be constructed through interaction with the construal, and these serve to identify potential computational agents and disclose a systematic framework within which they can act. In this way, as has been illustrated in a wide variety of examples [3–5], EM makes it possible to craft program-like behaviours out of initial exploratory interactions.

The primary tool that has been used to support EM to date - the EDEN interpreter - can be seen as generalising spreadsheet principles by supporting dependency maintenance on much richer data types. It has been used extensively in student project work over many years. By way of illustration, a Sudoku-solving construal developed using the web-enabled variant of the EDEN interpreter [3] can be accessed online. Making such a construal involves identifying 'all' the observables and dependencies that contribute to the human solver's perception of a concrete puzzle. These include attributes such as the location, size, background / foreground / border colour, current contents, status (i.e. whether given or to be determined), row / column / region number and index, potential contents etc for each cell of the grid. The spreadsheet-like dependencies that relate these observables are then recorded in a set of definitions. As an example, the plausible digits for cell 11 (as deduced from the simple criterion of not replicating a value that can already be found in the same row, column or region) are given by the definition:

```
possdig11 is all - (reg1digs + row1digs + col1digs)
```

where the four observables on the right-hand side are relational tables that respectively represent the set of digits $\{1, 2, ..., 9\}$ and the set of digits in the first region, row and column in the grid.

The construal can be exercised in several ways: as a learning environment for teaching solution techniques; as the basis for an environment for creating puzzles; to develop automatic solving procedures in which each step is explicable to a human solver. It is significant that writing a program to solve Sudoku puzzles does not necessarily shed light on any of these tasks, whilst making a construal helps to address all three. Interaction with the Sudoku-solving construal is open-ended in character: it can involve freely modifying dependencies and adding new ones. One extension of the construal we have explored ("colour Sudoku") involves making the background colour of a cell dependent on the set of plausible digits for that cell. One way to derive a program-like behaviour from the construal by exploratory interaction is then to redefine the value of the observable `all` so that a specific digit (say X) is eliminated from the set of admissible digits. By inspecting the impact of this redefinition on the background colours of cells, the human solver can infer the contents of cells to which simple rules such as "there is only one digit not already represented in the same row, column or region" or "there is only one cell in this row, column and region in which the digit X can be placed". This activity can then be automated, and reveals the potential and limitations of a solving strategy based solely on applying these simple rules.

Construals developed using EDEN reflect the exceedingly rich and potentially confusing nature of human cognition and agency, especially in unfamiliar contexts. Environment-agent EM is concerned with exploring the instrumental potential of devices, as illustrated by the new possibilities for solution that introducing "colour Sudoku" affords. But as far as making the transition from type C to type A modelling is concerned, the use of EDEN is problematic. Even for the Sudoku solving construal, more than five thousand observables are introduced. These observables take many different forms (display elements, tables, lists, scalars, strings etc), and linking them is syntactically complex. More complex aspects of the state of the construal have to be managed through importing files of definitions, and so may rely heavily but implicitly on appropriate configuration of the file system containing the construal. The modes of interaction with the construal are obscure to those unfamiliar with it, and have to be reinforced through regular practice. Whilst the flat space of observables may be well-matched to the Jamesian conception of "pure experience", there is no satisfactory way of devising, organising and recalling observable names. For all these reasons, any attempt to deploy EDEN in practical software development is confounded by the problems of scale and comprehension.

Despite these limitations, the EDEN interpreter has served to disclose potential for an EM approach to software development [2, 13] that has yet to be fully realised (cf. section 5). For instance, we have deployed a practical timetabling application based on EM [5], and shown in principle how formal systems can be conceived within the context of type C Modelling, and emerge from it. Experience of EM endorses the common sense idea that identifying reliable and stable patterns of agency and interaction in the application domain and empirically aligning the behaviour of software so as to accord with these patterns is prerequisite for successful formal specification. This is in keeping with the concern expressed by Jackson [10], echoing Vincenti [17], for formal specification to respect the need "to do justice to the incalculable complexity of the real-world".

4. Process-object EM for sw development

By comparison with a spreadsheet, the EDEN interpreter supports more extensive (and extendable) modes of perceptualisation, but it still relies on a repertoire of notations (for scalars, strings, lists, 2d points and lines, display elements such as panels and buttons, 3d graphics, geometric models, relational tables etc) which constrain the kinds of metaphor that can be invoked. The emergent structures and processes associated with an EDEN model can be exceedingly complex and subtle, but these are maintained in the mind of the modeller. As a result, EDEN has the same limitations as the spreadsheet in respect of scale and comprehensibility where authentic software development is concerned. This motivates an enhanced tool for EM that gives the explicit support for objects, structures and processes that is needed to smooth the transition from type C to type A modelling.

The DOSTE environment has been developed by the first author as an interactive tool that can enable concrete and exploratory modelling in the spirit of type C modelling. The conception of DOSTE draws on the object-oriented software development tradition, incorporating features similar to those invoked in Smalltalk, Self [16], GAUCHO [14] and SubText [7]. In its intended use, the focus is not however upon specifying abstract structures and processes, but upon realising the same kind of vivid live correspondence between a model and its referent that is illustrated in the spreadsheet-style development described above. The kernel of the model is the object structure as dynamically established by links between object components. The current disposition and the latent dependencies between these links are at all times explicit and open for reconfiguration by the modeller. It is the configuration of these links that is the counterpart of setting up the network of definitions within the spreadsheet: an activity that is informed by the need to maintain a semantic connection in immediate experience.

In a spreadsheet, the semantic force of a relation is not in the fact that there is an abstract relation 'a is b-c', but that 'profit is income - expenditure', and that all the interactions with the spreadsheet conform to and endorse this interpretation. The spreadsheet engine maintains dependencies between values whose meaningfulness resides in how they are manipulated by and mediated to the modeller rather than in their formal operational interpretation. In a similar spirit, the configuration of links is defined by dependencies that either serve to maintain "static" current relationships similar to those in a spreadsheet or to express the way in which a link will be updated from one moment by the next. Of course, the semantic significance of such links can only be apprehended through making their current state perceptible to the modeller at all times. To this end, within the DOSTE environment, the immediate structure of the object representation is made explicit to the modeller through maintaining a concrete perceptual representation of it.

The rethinking of software development that DOSTE affords was first motivated by designing a new kind of operating system. To date, perhaps its most significant application has been to computer games development. Whilst DOSTE draws inspiration from object-oriented software development ideas, it is entirely different in character from a conventional class-based object-oriented approach. As its associations with operating systems and computer games suggest, it is a framework for development that gives a high priority to concrete physical and experiential considerations. Whilst it creates explicit object and process structures, these are exceptionally fluid, and can be shaped as their perceptual counterparts are being experienced.

Our current research is focused on developing a new tool that integrates the DOSTE and EDEN environments, and in the process demonstrates the qualities of both environment-agent and process-object EM.

5. Our vision in summary

Existing modelling tools for software development tend to assume that we ultimately want conventional Turing-style programs to come out of it. They also assume that we have an understanding of the domain we are attempting to model. What these tools provide is a means of modelling a program to work out how to construct it

and to communicate this to others. In advocating type C modelling as a basis for software development, we look at things in a different way. Where appropriate, rather than modelling rule-based mechanisms, we model the environment in which such mechanisms are identified and interpreted. This has radical consequences both for the tools and thinking:

- **Challenging preconceptions about the need for paradigms and languages**. We do not regard it as essential to develop conventional programming language representations, and consider that dealing entirely with visual or other representations is acceptable and desirable. This poses fundamental issues relating to semantics and how meaning can be conveyed, but - drawing on James [11] - we are not daunted by these.
- **Accepting that a rigid program may not actually be the outcome**. We instead envisage the products of software development as constantly evolving and adapting to changing circumstances. Some software projects cannot reasonably be deemed to finish and to require a final state limits the kinds of software we can construct as well as how it can be used. Blurring the distinction between using a program and creating a program in the way we have discussed would enable software to constantly evolve in a changing environment.
- **Linking the construction of software to learning about the domain**. The central problem in software development is not how to make a program to perform some task but how to better understand that task and construct a model of it. By establishing an intimate connection between constructing software and gaining experience of the environment in which it operates, our approach enables one or more individuals to use the computer to learn about the problem domain.
- **Maintaining that abstraction is not necessarily required or desirable**. This is a key point. It seems that a lot of effort goes into adding ever increasing layers of abstraction to modelling tools and programming languages. We argue that what is actually required in many contexts is as little abstraction as possible and necessary. Abstraction distances the development from original domain of the problem and isolates the developer from their experience. By staying as close to the experienced problem as possible we can more easily construct, understand, alter and use the model.
- **Recognising the transformative role of dependency in programming-as-modelling**. The object-oriented paradigm has always aspired to establish a strong association between programming and modelling [6] - a vision that can be seen as quite different from that endorsed by the classical Turing view of programming. In our view, it is the introduction of *dependency* that has the most significant role to play in relating programming to modelling. What is more, the benefits of introducing dependency cannot be appreciated without the kind of radical shift in perspective that EM affords. As argued in [15], where the implications of adding dependency to the object-oriented Imagine Logo are considered, they cannot be realised by invoking dependency in an *ad hoc* fashion. This applies in particular to emerging practices in the software industry, such as the use of dependency injection and of Microsoft's implementation of .NET dependency properties in the Windows Presentation Foundation (WPF).

In pragmatic terms, we believe that our approach and orientation has wider applicability than is sometimes acknowledged. In the earliest stages of a project, where there is a typically poor understanding of the domain, it is best to remain as concrete as possible rather than start premature abstraction. As the model progresses it is possible that higher-level concepts and structures will emerge from the initial chaos. Gradually it may (but equally may not) be possible to come up with a complete and formally verifiable model - if that is the aim.

From a more philosophical perspective, it is fashionable in some circles to conceive the world as a computational process. Whether or not we subscribe to this view, not all human experience of the world has the same character as participation in a rule-based activity. If there *are* rules that govern everything we observe, we are not necessarily aware of them. *Software for humanity* has to be developed with this fact in mind.

Acknowledgments

We are much indebted to Steve Russ, to the many student contributors to Empirical Modelling research, past and present, and especially to Richard Cartwright, Pi-Hwa Sun, Russell Boyatt, Ashley Ward, Antony Harfield, Allan Wong, Zhan En Chan and Richard Myers for their groundwork on tool development.

References

[1] Empirical Modelling. `http://www.dcs.warwick.ac.uk/modelling` [Accessed 13/08/10].

[2] M. Beynon, R. Boyatt, and Z. E. Chan. Intuition in Software Development Revisited. In *Proceedings of 20th Annual Psychology of Programming Interest Group Conference*, 2008.

[3] M. Beynon, R. Myers, A. Harfield, and S. Russ. The Sudoku Experience. `http://www.dcs.warwick.ac.uk/~wmb/sudokuExperience/workshops/` [Accessed 13/08/10].

[4] W. M. Beynon, J. Rungrattanaubol, and J. Sinclair. Formal Specification from an Observation-Oriented Perspective. *Journal of Universal Computer Science*, 6(4):407–421, 1999.

[5] W. M. Beynon, A. Ward, S. Maad, A. Wong, S. Rasmequan, and S. Russ. The Temposcope: a Computer Instrument for the Idealist Timetabler. In *Proceedings of the 3rd international conference on the practice and Theory of Automated Timetabling*, pages 153–175, August 2000.

[6] G. M. Birtwhistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Chartwell-Bratt Ltd, 1979.

[7] J. Edwards. Subtext: Uncovering the Simplicity of Programming. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 505–518, 2005.

[8] D. C. Gooding. *Experiment and the Making of Meaning*. Kluwer Academic Publishers, 1990.

[9] D. Harel. Biting the Silver Bullet - Toward a Brighter Future for System Development. *IEEE Computer*, 25(1):8–20, 1992.

[10] M. Jackson. What Can We Expect From Program Verification? *IEEE Computer*, 39(10):53–59, Oct. 2006.

[11] W. James. *Essays in Radical Empiricism*. Longmans Green, 1912.

[12] B. Nardi. *A Small Matter of Programming*. MIT Press, 1993.

[13] P. Naur. Intuition in Software Development. In *TAPSOFT*, volume 2, pages 60–79, 1985.

[14] F. Olivero, M. Lanza, and M. Lungu. Gaucho: From Integrated Development Environments to Direct Manipulation Environments. FlexiTools Workshop, May 2010.

[15] C. Roe and M. Beynon. Dependency by definition in Imagine-d Logo: applications and implications. In *Proc. of the 11th European Logo Conference*, Bratislava, August 2007.

[16] R. B. Smith, A. B. Smith, and D. Ungar. Programming as an Experience: The Inspiration for Self. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 303–330, arhus, Denmark, 1995. Springer-Verlag.

[17] W. G. Vincenti. *What Engineers Know and How They Know It*. The Johns Hopkins University Press, Baltimore, 1993.