Modelling Software Organisations

David Hales
Centre for Policy Modelling, Manchester Metropolitan University. UK.
dave@davidhales.com

Chris Douce
Feedback Instruments, Crowborough, UK.
chrisd@fdbk.co.uk

Keywords: POP-I A, group dynamics, team structure, POP-V B, research methodology.

Abstract

We view software construction as a *social process* embedded within organisational and cultural structures. To increase our understanding of the process of software construction we attempt to model not just individuals or teams but organisational wide processes. How can we begin to model such complex dynamic entitles composed of many interacting individuals? We outline our initial steps towards such an enterprise and argue that this is a worthwhile endeavour if we are to increase our understanding of software production in the real world[1].

1. Introduction

We aim to increase the understanding of the process of software construction, approaching this problem from a social (organisational)

[1] We want to make explicit that our goal is *not* to produce models that *replace* the function of a software organisation or developers. We are not here concerned with the *actual* construction, specification or delivery of any *real software systems*. This would be in the realm of software engineering and formal specification languages. We are proposing highly abstract models of some aspects of social processes related to software development within organisations.

perspective. It would seem that all software construction (certainly at a large-scale) is fundamentally a social process. Programmers work within teams and groups which themselves are embedded within larger organisational structures and cultures. These social structures enable, constrain and shape the behaviour, knowledge and general conceptual repertoires of individuals involved in the software construction process.

The process is governed by and situated within complex social institutions (whether traditional commercially driven hierarchical corporations or decentralised electronically mediated cooperative non-commercial enterprise). It may even be claimed that it is the organisation and its culture that are even more important in determining the quality and efficiency with which software is produced rather than the individuals that comprise it.

A method is outlined by which we intend to explore these organisation wide phenomena in a quasi-empirical way. We intend to use empirical work carried out within the Psychology of Programming (PoP) and apply it to the construction of an Agent Based Social Simulation model (ABSS). Construction of such a model will force us to be explicit about our assumptions and highlight gaps in our understanding and suggest novel hypotheses and future empirical work. Experimentation using the model will allow for the comparison of various organisation structures, cultures and models of programmer cognition.

Firstly, we briefly outline the ABSS approach then we give an overview of some of the relevant PoP literature. We discuss an initial model ontology, based on the concept of "informational artefacts", and briefly describe how different kinds of organisational structures and cultural phenomena might be captured by such an ontology. Finally we summarise our approach as a complementary guiding method that can be combined with and profit from additional empirical investigation.

2. Agent Based Social Simulation

Agent Based Social Simulation (Moss & Davidsson 2000)[2] is an approach to modelling of social organisations based on the specification of individual agents and the analysis of the resulting emergent social

[2] The ABSS community organise a Special Interest Group (ABSS-SIG) within the E.U. AgentLink II Network (see http://www.agentlink.org/activities/sigs/sig4.html).

phenomena that arise from their interactions over time. Roughly two kinds of approaches are used in this area. One approach, which might be termed the *artificial* approach, explores the behaviour of interacting agents with artificial worlds. The implementation of agent cognition is often based on intuition and rarely grounded in any actual empirically based theory or evidence. These kinds of exploratory models are sometimes termed "artificial societies" and have some similarity with the models produced within "artificial life". Another approach attempts to ground agent cognition, behaviours and interaction scenarios on empirical observations[3].

This latter approach is increasingly being applied to real world scenarios in order to gain a deeper understanding of the kinds of complex interactions shape real social phenomena[4]. The major use of such models isn't prediction in any quantitative sense (the complexity and contingency of most social phenomena make this a futile exercise) but to gain a deeper qualitative understanding of the process dynamics. More importantly, to produce may "what if" runs of models in which different conditions, events and model components can be compared (Carley 1998, Terán et al 2000).

What appears to be emerging from attempts to ground models in empirical observation is the need for scenario specific detail. Just how do agents in given scenarios perceive their roles, formulate their goals, interact with others and behave. It seems that no general theory can provide these answers. The only way forward therefore is the detailed study of the actual phenomena of interest. It would seem that some of the existing body of empirically based work that has been carried out in Psychology of Programming studies could be of great use for informing an agent based model of the process of software construction as *a social process within organisations.*

[3] See the Journal of Artificial Societies and Social Simulation (JASSS) for examples of both kinds of model. The Journal is available free at: http://www.soc.surrey.ac.uk/JASSS

[4] For example, the on-going European wide FIRMA project on water usage. See http://firma.cfpm.org/ for details.

3. Models of Programming

Software comprehension and understanding models found within the psychology of programming have commonly focussed upon the individual and their abilities (Shneiderman & Carrol, 1988), the type of information that dominate program comprehension (Pennington, 1987) and models of how strategy can be affected by domain knowledge. There are fewer models of software production or generation, some of which are described by Davies (1993).

Another aspect to understanding the programmer relates to understanding the environment and interactions that can occur between other programmers. This is evident in the current enthusiasm regarding the 'eXtreme' programming notions, where issues such as "pair-programming", where two programmers work closely together on one programming problem, are discussed (Beck, 1999). It is hoped that two programmers produce a more comprehensible, robust software system within reduced timescales.

Two programmers working together on a project are more often than not a part of a larger team within an existing organisation. Several studies have been performed to study and attempt to understand how teams of developers function. An example being within the work of Curtis (1988) which discusses programming knowledge in terms of individual, team, project, company and business milieu.

One of the legendry stories associated to the conception of a 'programming group' describes a situation where a coffee machine situated in a key location within an organization was removed. The coffee machine created a place where software developers could informally meet and discuss issues and trade techniques. The removal of the coffee machine would then result in a lack of communication between key developers that would potentially impact on the performance of a group of developers.

Other literature that is important relates to the cognition of design, and approaches to the construction of teams to support effective design. Within the literature, some work has been performed to study design in an ecological environment, namely, the situations and environment where software development takes place (Visser, 1990).

Returning to the individual programmer, to construct and comprehend software, a programmer is required to perform logical reasoning, using short-term memory to encode and store information obtained from a variety of sources. One approach used to study logical reasoning within Cognitive Science has been to construct software models that aim to simulate logical reasoning, a technique not unlike the one that is being described. The behaviour and performance of a software system can be compared with the behaviour of a subject who has carried out similar tasks within the laboratory. Differences between the two can be used to provide evidence in support of a particular model of reasoning that has been essentially formalised in software code.

Understanding and construction of models within the Psychology of Programming can occur through two routes. One approach is the empirical route. Programmers and development teams can be studied both inside and outside the laboratory. Evidence of behaviour can be collected which can then assist in the construction and development of new or existing models, or through the proving or disproving or hypotheses. The second approach could be called the traditional engineering method, where researchers and developers attempt to produce artefacts that are intended to develop and enhance the programmers working environment. Through the application of a new tool or technique, new phenomena or problems may be discovered. This may provide an insight into the needs of the community where the tool is used, suggesting further enhancements and improvements that could be made. Subsequent development incorporates proposed enhancements allowing the process to continue.

The modelling of a software organisation is primarily an engineering activity. Models have to be designed and constructed. It is also an empirical process, since models that are built cannot easily be considered to be 'correct' in the software engineering sense. A model may exhibit wide variability in its behaviour based upon the parameters and its architecture.

Within the modelling community, models may be proposed and implemented. Results from models would be captured, analysed and then published. A result from one model is entirely at the mercy of its implementation, primarily since first implementation of a model may have guided its design. Since details of the model are published, just like

any other empirical techniques, they can be subjected to scrutiny. Models can be evaluated and reported results potentially replicated. The results would then either weaken or strengthen the validity of a model. In cases where the results are different, this may indicate that the model that has been described is not directly comparable.

4. Information Artefacts[5]

A general rule-of-thumb for any modelling enterprise is to construct a model that is "as simple as possible but no simpler". In this spirit initial thoughts on the way to capture the processes involved with software construction in the simplest possible way without losing sight of any phenomena that may be of interest are presented.

An organisation can be viewed as collection of agents, interaction possibilities and informational artefacts. By agents we mean individuals or possibly teams with some set of core skills or competences and some ability to learn and adapt. The relationships between the agents determine the interaction possibilities – for example, a manager may co-ordinate with other manager agents and delegate to lower level operative agents.

By informational artefact we mean to capture a whole raft of phenomena in a single abstraction. Effectively, we wish to label anything within an organisation that directly affects the performance of an agent in achieving its goals (but that is not another agent or a relationship between agents) as an informational artefact. By this we mean things such as documentation, methods of coding, fixes and workarounds that can be communicated between agents either formally via persistent documentation or informally via advice, guidance or habitual work patterns.

For agents to perform tasks (such as programming) they need to combine intrinsic skills with the correct set of informational artefacts. More precisely, given an agent with some task T to perform, the task may require less effort (time) given some associated informational resource (artefact). However, it will also require effort to locate the resource or even to ascertain if such a resource exists.

---

[5] This concept of "informational artefacts" is loosely based on Green (2000) although as presented here probably has more in common with the area known as "memetics" (for example see Speel H-C 1997).

In this context of the proverbial coffee machine, this can be viewed as an interaction catalyst that facilitates the interchange, trading and replication of valuable informational artefacts. The moral of the story is that the informal nature of the interactions (around coffee breaks) provides an ideal setting for the exchange of informal artefacts. Such informal knowledge is just as important (if not more so) than formal documentation or procedures.

An idealised hierarchical model of a software organisation might work by submitting a high level task in the form of a set of requirements. The task would be decomposed and passed to agents lower in the hierarchy. This process would continue until a level was reached at which agents with the intrinsic skills and access to the required informational artefacts could complete the task.

Alternatively, consider a more defuse and diverse bottom-up organisation. Here no overall high-level task (or goal) would be submitted at all, but individual agents at the lowest level would collaborate and work on tasks suited to their skills and situation. From this a higher-level goal or task may emerge. Agents identifying the higher-level task may become responsible for bringing together the necessary artefacts to complete the task. The final result of such a process is unpredictable yet highly productive.

In a model of a software organisation we envisage a model in where individual tasks may be represented as some set of skill requirements and time based effort requirements (these could be specified as simply as pairs of single values). Agents may have diverse and/or multiple skills. Additionally informational artefacts may be represented as pairs of values indicating associated skill and effort reduction factor. So an agent may complete a task more quickly given access to a relevant informational artefact. Without such an artefact the task may still be completed but with more effort (time). So artefacts may be stored independently of agents, others may only survive by repeated replication through agents' memories.

Some artefacts may be so prevalent and permanent as to become 'ambient'. Examples of ambient information artefacts might be software tools, methodologies version control and configuration management systems. Every developer must have a common appreciation of such tools since they are pervasive throughout the organisation.

In the simplest form, a model could boil-down to sets of value pairs used to represent required skills and effort. Within a model, a specification could be decomposed, relevant skilled agents located and informational artefacts utilised. Particular kinds of organisation structures and individual agent characteristics would affect the success and performance of the organisation in completing the task. The model could be refined to capture more details of a task by increasing the expressive power of the model by using more than simple pairs of values.

5. Summary

It is proposed that computational modelling of organisations could be used as an exploratory technique to study the activities of software developers. The process of constructing a model requires its designer to formalise many potentially intangible processes that occur and concepts that may be used during the software development. The process of considering each element forces the modeller to address these issues in depth, which may potentially necessitate investigation in areas that may not have previously been considered.

For the software development community, the purpose of studying the modelling of software development should be clear. Experimenting with real groups of programmers, however ecologically valid, is incredibly expensive necessitating laboratories, recording equipment, experienced software developers and projects, see Shneiderman and Carrol (1988). Experimenting, however abstract, with a computer-based interactive model is substantially cheaper but may result in behaviour that may be somewhat removed from the real programming laboratory.

Modelling of software development activities and studying of programmers within a laboratory should be considered as two complimentary empirical approaches, where each technique could suggest or provide a guiding direction to the other.

6. References

Beck, K. (1999) Extreme Programming Explained. Addison Wesley.

Carley, K. M. (1998) Organization Adaptation. Annals of Operations Research, vol. 75, 25-47.

Curtis, B. (1988) A field study of the software design process on large systems. Communication of the ACM, vol. 13, no. 11, 1268-1287.

Davies, S. P. (1993) Models and theories of programming strategy. International Journal of Man-Machine Studies, vol. 39, 237-267.

Green T. R. G. (2000) Instructions and Descriptions: some cognitive aspects of programming and similar activities. Invited paper, in Di Gesu., V., Leviadi, S. and Tarantino, L., (eds.) Proceedings of Working Conference on Advanced Visual Interfaces (AVI 2000). New York: ACM Press, pp21-29. Available at: http://www.ndirect.co.uk/~thomas.green/workStuff/papers/

Moss, S., Davidsson, P. (2000) Multi-Agent-Based Simulation. Lecture Notes in Artificial Intelligence 1979. Berlin: Springer-Verlag.

Pennington, N. (1987) Stimulus structures and mental representations in expert comprehension of computer programs. Cognitive Psychology, vol 19, 295-341.

Rist, R. S. (1989) Schema Creation in Programming. Cognitive Science, vol 13, 389-414.

Rogalski, J. & Samurcay, R. (1990) Acquisition of programming knowledge and skills in Psychology of Programming, eds: Hoc, J.-M., Green, T. R. G., Samurcay, R & Gilmore, D. J. Academic Press.

Shneiderman, B. & Carrol, J. M. (1988) Ecological Studies of professional programming. Communications of the ACM, vol 31, 1256-1258.

Shneiderman, B. & Mayer, R. E. (1979) Syntactic/semantic interactions in programmer behaviour: a model and experimental results. International Journal of Computer and Information Sciences, vol 8, 219-238.

Speel, H-C., 1997; A Memetic Analysis of Policy Making. Journal of Memetics - Evolutionary Models of Information Transmission, 1(2).

Available at: http://jom-emit.cfpm.org/1997/vol1/speel_h-c.html

Terán, O., Edmonds, B., and Wallis, S. (2000) Mapping the Envelope of Social Simulation Trajectories. In Moss, S., Davidsson, P. (Eds.) Multi-Agent-Based Simulation. Lecture Notes in Artificial Intelligence 1979. Berlin: Springer-Verlag (available at: http://cfpm.org/cpmrep72.html).

Visser, W. & Hoc, J.-M. (1990) Expert software design strategies in Psychology of Programming, eds: Hoc, J.-M., Green, T. R. G., Samurcay, R & Gilmore, D. J. Academic Press.