The Impact of Using Pair Programming on System Evolution: a Simulation-Based Study

Paul Wernick and Tracy Hall
Systems and Software Group, School of Computer Science
University of Hertfordshire
College Lane, Hatfield, Hertfordshire AL10 9AB, England
tel. ++1707 286323/284782; fax ++1707 284303
{p.d.wernick, t.hall}@herts.ac.uk

Abstract

In this paper we investigate the impact of pair programming on the long term evolution of software systems. We use system dynamics to build simulation models which predict the trend in system growth with and without pair programming. Initial results suggest that the extra effort needed for two people to code together may generate sufficient benefit to justify pair programming.

Keywords

software evolution, simulation, system dynamics, software process, agile, pair programming

1. Background and context

As software systems become pervasive it is increasingly important to manage their evolution over the many releases of their long-term useful lives. Being able to predict the growth of a software system over long periods of time will allow long-term planning of the process which will evolve that system. This benefits both those who develop and evolve the system, and the stakeholders of the system.

The term 'agile' describes a collection of development approaches, one of the best known being eXtreme Programming [2]. This approach embodies a set of software development practices which includes pair programming. It has been suggested that some of these practices might successfully be applied to otherwise-unchanged 'traditional' software processes [14].

Software development and process change are often implemented before their long-term implications have been determined. A valuable means for examining such long-term implications is through the use of simulation studies. We have previously used System Dynamics [6] (SD) simulation-based studies to investigate the causes of long term software evolution trends [12. We present a simple SD model to show the potential effects on long-term software system growth of adding pair programming to a traditional software process. The model tracks changes in numbers of requirements met over time by a software product, comparing trends with and without pair programming.

The research question we address here is, therefore: *how does the use of pair programming impact on the long- term trend in software product evolution?*

Our ability to answer this question is based on the simulation model which we present, and the validity of the calibration parameters which we have obtained from our previous work and from the literature. The answer obtained in our work relates to the numbers of requirements which can be implemented and delivered to system users over many releases of a software product.

2. Rationale and approach to research

2.1. Pair programming

Agile approaches have emerged in direct response to the reported poor performance of plan-driven approaches.[*] A variety of case studies have been reported which suggest that the use of agile approaches deliver software with increased user satisfaction, lower defect rates and increased productivity [3, 5]. A well-known example of such a process is eXtreme Programming (XP) [2].

Pair programming is one of the core practices of XP. "All production code is written with two people looking at one machine, with one keyboard and one mouse. … There are two roles in each pair. One partner, the one with the keyboard and the mouse, is thinking about the best way to implement this method right here. The other partner is thinking more strategically" [2, p.58]. Pair programming has been selected as the subject of this simulation exercise because it is a separately identifiable practice within XP which can be used

[*] 'Plan-driven' is the term coined by Barry Beohm (2003) to describe all non-agile development approaches.

within an otherwise-unchanged existing software process; indeed it is already being employed in such a way (“Developers are trying pair programming without any other agile practise …” [15]). In addition, numerical data on pair programming is available from the literature which enables model calibration.

Pair programming is claimed to improve code quality, to allow both system-level and code-level issues to be considered simultaneously, and to help maintain code quality and standards [2, p.100–102]. Pair programming results in code which is understandable by both members of the pair, rather than by a single author. The result of this should be code that is easier to understand and thus to maintain. The use of pair programming should, therefore, have an observable impact on system evolution.

However, the rate of output of lines of code for a pair is somewhat less than for two individual programmers working alone. Williams *et al.* [15] have reported a reduction in overall output of 15%.

2.2. Simulation-based studies

A simulation-based approach offers advantages over experimental or observational studies. In particular, simulation can enable the assessment in advance of long-term process behaviour, and the effects of proposed process changes. In the case presented here, simulation allows the assessment of the long-term effects of incorporating a specific practice into an otherwise-unchanged software process. Such behaviour could otherwise only be examined in the real world, by the *post facto* examination of results of long-term metrics collection programmes.

Our approach uses metrics collection and simulation-based studies in combination. Metrics derived from shorter-term studies of actual practice are used in calibrating simulations of longer-term evolution processes. This approach will help identify those process improvements which have the greatest benefits for the long-term evolution of a software system.

Our approach is firstly to develop a ‘base’ simulation of a long-term software development/evolution process. We then add to this model the simulated effects of the practice(s) under investigation, and compare the model outputs with and without this. This approach has previously been used by Tveldt and Collofello [10] and Haberlein [7].

Our base model is designed to be as simple as possible. This approach continues that adopted in our earlier studies [4, 12, 13]. It contrasts with the more complex base model employed by Tveldt and Collofello and by Haberlein, *viz.* that of Abdel-Hamid and Madnick [1].

2.3. System dynamics introduced

The ubiquity of complex inter-relationships in the real world makes it difficult to isolate specific aspects of systems for investigation by conventional statistical analysis. The aim of SD is to represent the complexities of real world situations in a dynamic simulation model.

SD modelling is based on the concept of a hydraulic system of 'stocks' and 'flows'. All elements of a system are considered in terms of these concepts. For example, software requirements can be considered as a stock that flows from users to analysts. Time delays that in practice slow this flow also need to be taken into account, since they result in many of the behaviours observed in real-world dynamic systems. For example, the generation of new requirements after a system has been fielded takes time, as the users familiarise themselves with the new system and realise that changes can usefully be made to it.

Quantitative SD models are based not on traditional statistical correlations but on simulating over time the dynamic interactions of information flows. These interactions are captured in causal diagrams, and their quantitative results calculated by mathematical integration over time of rates of flow of artefacts and control information around the model.

3. The base model described

3.1. Model structure

The model presented in this section represents the structures, effects, inputs and outputs of a long-term software evolution process. To allow the simulation of the effects of adopting new methods and approaches, it is important that the base model is simple. A simple model will allow the simulation of generalised software development and evolution activities without any bias for or against any particular method, toolset or approach, and reduces both the number of assumptions needing to be made about the process and the number of calibration inputs required.

High-level dynamic behaviours resulting from outer feedback loops can outweigh more local feedback effects from inner loops in determining system behaviour over time. By concentrating effort on modelling the most significant, outer, feedback loops in a process, more detailed process structure can be abstracted, and the need to collect more detailed data avoided. Such a model can still provide useful insights despite high levels of abstraction.

Our base model is shown in Figure 3. It builds on ideas, structures and values in our previous software evolution simulation models. In particular, it incorporates feedback structures representing both the generation of new requirements [4] and the correction of faults in previously-implemented requirements [13].

The model works as follows. The software development process is viewed as a mechanism to convert ‘requirements which need to be met’ into ‘requirements which have been met and fielded to users’. The computed rate of software development is a function of the effective human resource available to perform that work; other relevant factors such as training, motivation and tool support (with the exception of the inertia effects of the existing system which is being modified) are considered in the model as factors which change the value of this factor. This starting rate is subjected to a time delay function to represent the time taken to perform the development work. It is further delayed as completed requirements have to wait until the next release of the software is delivered to its users. The effective effort available is reduced over time [12] due to the need to make new functions fit into the existing system, in a manner related to the existing system size [11].

As a result of fielding the software, two things happen. First, users of the system tell the developers that some requirements have not been met properly, typically due to bugs in the fielded software and/or mistaken interpretation of the users' requirements by the developers (cf. [13]). Second, system users identify new requirements due to additional ways of using the system made possible by newly-fielded software enhancements (cf. [4]). In addition, exogenous events can arise, i.e. changes in the environment within which the system is used and for which the system has to be modified. For the model calibration, the value for exogenous events has been set to 0, but an input for these has been included in the model for completeness.

3.2. Model Calibration

The calibration inputs for the evolutionary growth described in the model are based on actual figures for the evolution of the VME mainframe operating system as described in [4]. This work on VME is useful to our model building as both input values and outputs against which to check them are available. The time delays used in the model are averages of the variable delays used in the VME model. These delays are

- time taken for the conversion of requirements into code ready to be released: 8 months;
- delay from the completion of this until next release is delivered to users: 5 months; and
- delay for user adoption of the new release, and for the feeding back of new requirements or of errors requiring fixing: 8 months.

In the absence of actual data:

- system size at the start of the simulation run is set to 200 requirements units;
- initial input workload of recognised but unfulfilled demand for new requirements is set to 50 units;
- input value of effort available to turn requirements into met requirements is set to a constant value of 1 unit per month, reflecting the lack of change in VME kernel team size over the time simulated in the base model.

For the VME model the flow of completed requirements was multiplied by a factor of 0.6 to generate the flow of new requirements. In the model described here, we have multiplied the output of the software production process delay function by 0.64 for the generation of new requirements and 0.16 to produce the rate of error feedback. The ratio of new to incorrect requirements is thus 4:1 in accordance with Pressman's [9, p.849] conclusion that fixing mistakes comprises 20% of maintenance work, the other 80% being system adaptation and enhancement for users or future use.

The input effort value is multiplied by a factor reflecting the effects of the existing system size on the ability to evolve it. This results in a reduction of effective throughput over time, due both to the inertia of the existing system and to a reduction in system-wide knowledge [12]. The overall effect on effort due to existing system size is calculated as a multiple of the inverse cube [11] of that size.

Finally, the simulated time over which the model runs is 156 months, reflecting the period of time over which the VME data is available.

3.3. Calibrating model output to real-world values

It is necessary to select one amongst the many real-world product trends for the simulation to follow. These product metrics include physical system size (cf. [4]) and the number of units of requirements implemented and delivered (cf. [13]). We decided to use the latter, since it is a more direct reflection of the ability of a system to do useful work by meeting its users' needs. It also avoids issues related to code size measurement, and the need to reflect the effect on user satisfaction of exogenous events. Situations in which delivered functionality needs rework are captured by the feedback loop of requirements not met correctly in Figure 3.

The output trend for requirements met and fielded is shown in Figure 1. This shows typical growth in product size over time, reflecting a reduction over time in the implementation rate of new requirements (cf. [4], 13]). An equivalent trend for actual growth in VME from the same arbitrary starting point, in terms of numbers of management-level requirements units multiplied by their average size in modules, is shown on the same axes in Figure 1. The smooth trend of the simulation output contrasts with the less consistent actual software product evolution trend, due to the abstraction in the model of detail in the real-world software evolution process which causes higher-order dynamic behaviour.

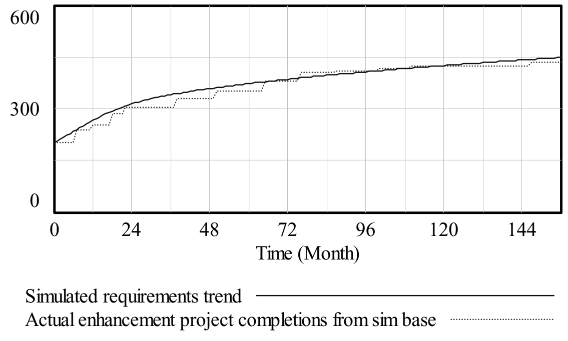

Figure 1: Requirements met over time by the 'typical' software evolution process

4. Adding pair programming to the model

No changes to the base model structure have been required to simulate the effects of using pair programming. Additional variables have been added to allow the quantitative effects of process changes such as the introduction of pair programming to be reflected in simulation runs, these are shown in capital letters in Figure 3. These variables have been identified from the claimed benefits of pair programming as discussed above.

Values for model parameter values to simulate the effects of pair programming have been taken in the main from literature; these could be replaced by actual values where available when the model is calibrated in practice. The values used here have been derived as follows:

- *development time scaling factor:* reduced by 40% [8, p.22]; [15, p.19]. A reduction in cycle time due to the adoption of pair programming only affects the time to develop the software, not the time the software has to wait until the next release. The interval between releases is therefore only shortened by the saving in *development* time due to pair programming
- *input effort scaling factor:* effective effort is reduced by 15% [8, p.22]; [15, p.19].
- *fault generation scaling factor:* based on Williams *et* al's [15] observation of increased quality in terms of the relative numbers of programming faults in developed software. Their results indicate a reduction in errors to 300/356.5 of the value without pair programming. This is used to reduce the rate of user-reported analysis faults generated in the simulation.
- *inertia scaling factor*: reflecting improved long-term system maintainability due to better quality/more understandable code and more knowledge of the system. As far as we know, no research into long-term benefits of pair programming for ease of system maintenance has been published. We have therefore estimated this value. Model runs have been undertaken using a 5% reduction in system evolution effort as a conservative estimate and 40% as a high estimate.

Other parameters are unchanged from the base calibration.

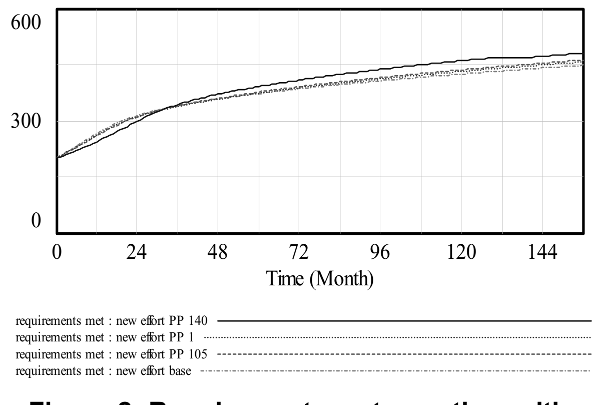

Figure 2: Requirements met over time with and without pair programming

Running the simulation with and without pair programming suggests that the results of implementing pair programming will be positive. Although pair programming does cost more, the gains appear to outweigh this, providing an overall benefit. The degree of benefit depends on the potential for pair programming to improve the long-term maintainability of the system, which is as yet undetermined.

Figure 2 shows the results of simulation runs with zero (PP 1), 5% (PP 105) and 40% (PP 140) gains in maintainability plotted on the same axes together with the base model output. The results obtained suggest that the adoption of pair programming results in a gain in delivered system functionality over time, the actual improvement in process performance depending on the degree of improvement in system evolvability.

The trends shown in Figure 2 also demonstrate that any improvement in process performance from the adoption of pair programming can be considered from two evolutionary viewpoints. Firstly, the process provides an opportunity for greater long-term system growth and thus system longevity. Secondly, it means that any particular level of functionality will be delivered to users sooner. For example, at month 120, a 40% gain in maintainability results in 461 units of requirement being delivered as against 421 for the non-pair programming case. Alternatively, the same pair programming case delivers 420 units of requirement in month 81, compared with month 119 for the non-pair programming case.

These results assume that users will adopt, use and provide feedback from any increase in fielded functionality.

5. Conclusions and future work

The simulation-based study presented here has allowed us to examine the implications of a suggested process improvement on a longer time scale than is possible for a real-world study of a comparatively newly-suggested process change. Our work suggests that pair programming is likely to produce long-term software evolution benefits, the exact value of which depends on any improvement in maintainability of pair programmed code. There is no evidence currently available as to whether two people cooperating to write code produce code which is more maintainable in the long term. Consequently no well-quantified answer is available to our initial research question, and more work will need to be done to measure the impact of pair programming on long-term evolution.

Our approach assumes that it is possible to inject one new practice such as pair programming into an existing long-term software evolution process without affecting other aspects of the process. Our current simple model does not reflect these possible knock-on effects. In addition, our simulation reflects the effect of one process change *viz.* that of adopting pair programming applied to an otherwise-unchanged process. As Paulk notes, there are "strong dependencies between many XP practices ..." [8, p.23]. These interactions will need to be taken into account if more than one practice is introduced simultaneously; the effects of each change cannot simply be added or multiplied to predict the combined effect.

Future work on this model is likely to include the simulation of other XP practices. Modifying the base model to use switch variables to turn on and off the simulated effect of a new practice will make comparative simulations easier to undertake. It is also hoped that the development and calibration of this model will continue with industrial collaborators using agile processes in a commercial environment. Such work will also allow the replacement of 'general-purpose' values taken from the literature for use as parameter values in our model with actual figures taken from the specific experience of industrial developers whose process behaviours we are to simulate. This will also allow, indeed require, the model to be calibrated against other actual processes and product metrics other than delivered requirements, and thus increase our confidence in its validity and predictions.

Most importantly, our work reveals that an improved understanding of long-term maintainability of software is vital to predicting the impact of any process change. This understanding must embrace the relevant mechanisms within the global software process as well as technical issues.

References

[1] T.K. Abdel-Hamid and S.E. Madnick, *Software Project Dynamics – An Integrated Approach*, Prentice-Hall, New Jersey, 1991.

[2] K. Beck, *Extreme Programming Explained*, Addison Wesley, Boston, MA, 2000.

[3] B. Boehm and R. Turner, "Using risk to balance agile and plan-driven methods", *IEEE Computer*, 2003, vol **36**, pp.57–66.

[4] B.W. Chatters, M.M. Lehman, J.F. Ramil and P. Wernick, "Modelling A Software Evolution Process", *Software Process: Improvement and Practice*, 2000, **5** (2–3), pp.91–102.

[5] M. Cohn and D. Ford, "Introducing an agile process to an organization", *IEEE Computer*, 2003, **36** (6), pp.74–78

[6] J.W. Forrester (1961) *Industrial Dynamics*, Productivity Press, Cambridge, MA.

[7] T. Haberlein, "A Framework for System Dynamic Models of Software Acquisition Projects", 2003, *Proc. ProSim 2003*. Portland, OR.

[8] M.C. Paulk, "Extreme Programming from a CMM Perspective", *IEEE Software*, November/December 2001, **18** (6), pp.19–26.

[9] R.S. Pressman, *Software Engineering*; European Adaptation by D. Ince, Addison Wesley, 2000.

[10] J.D. Tveldt and J.S. Collofello, "Evaluating the Effectiveness of Process Improvements on Software Development Life Cycle Time via System Dynamic Modelling", 1995, *Proc. COMPSAC '95*, pp.318–325.

[11] W.L. Turski, "The Reference Model for Smooth Growth of Software Systems Revisited", *IEEE Trans. Software Engineering*, 2002, **28** (8): pp.814 – 815.

[12] P. Wernick and T. Hall, "Simulating Global Software Evolution Processes by Combining Simple Models: An Initial Study", *Software Process: Improvement and Practice*, 2002, **7** , pp.113–126.

[13] P. Wernick and M.M. Lehman, "Software Process Dynamic Modelling for FEAST/1", *J. Systems and Software*, 1999, **46** (2/3), pp.193–202.

[14] L. Williams and A. Cockburn, "Agile software development: It's about feedback and change", *IEEE Computer*, June 2003; pp.39–43.

[15] L. Williams, R.R. Kessler, W. Cunningham and R. Jeffries, "Strengthening the Case for Pair Programming", *IEEE Software*, July/Aug. 2000, **17**, 4, pp.19–25.

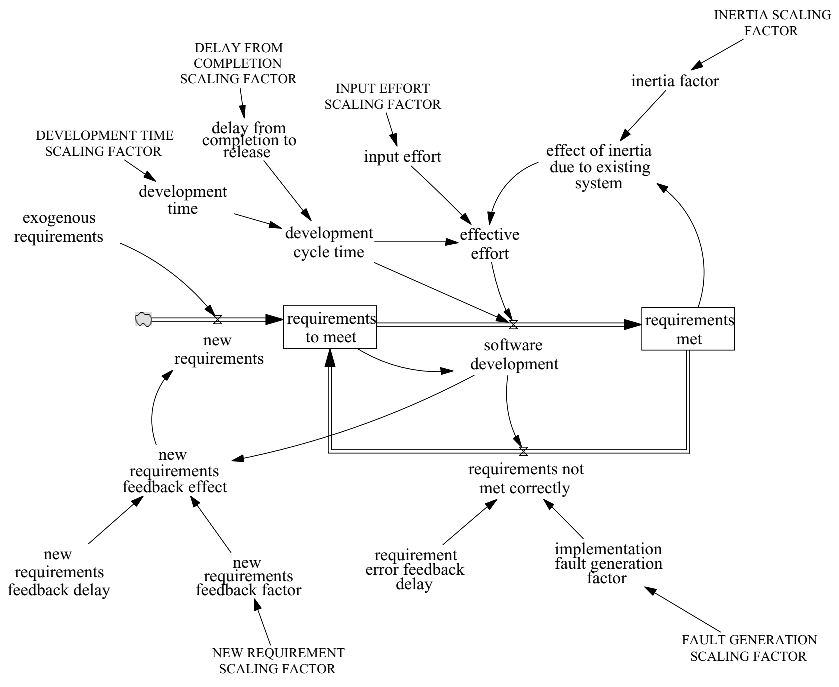
INERTIA SCALING FACTOR
DELAY FROM COMPLETION SCALING FACTOR
INPUT EFFORT SCALING FACTOR
inertia factor
DEVELOPMENT TIME SCALING FACTOR
delay from completion to release
input effort
effect of inertia due to existing system
development time
exogenous requirements
development cycle time
effective effort
requirements to meet
requirements met
new requirements
software development
new requirements feedback effect
requirements not met correctly
new requirements feedback delay
new requirements feedback factor
requirement error feedback delay
implementation fault generation factor
NEW REQUIREMENT SCALING FACTOR
FAULT GENERATION SCALING FACTOR

Figure 3: Simulation of 'typical' software evolution process