An Empirical Study of a Software Maintenance Process

R Harrison, R Nithi, K T Phalp, L G Samaraweera & A P Smith

Department of Electronics and Computer Science,
University of Southampton,
Southampton, SO17 1BJ, UK.
Fax. +44 (0) 1703 593045

Abstract. This paper describes how a process support tool is used to collect metrics about a major upgrade to our own electronic retail system. An incremental prototyping lifecycle is adopted in which each increment is categorised by an effort type and a project component. Effort types are *Acquire, Build, Comprehend and Design* and span all phases of development. Project components include data models and process models expressed in an OO modelling language and process algebra respectively as well as C++ classes and function templates and build components including source files and data files. This categorisation is independent of incremental prototyping and equally applicable to other software lifecycles. The process support tool (PWI) is responsible for ensuring the consistency between the models and the C++ source. It also supports the interaction between multiple developers and multiple metric-collectors. The first two releases of the retailing software are available for *ftp* from *oracle.ecs.soton.ac.uk* in directory *pub/peter*. Readers are invited to use the software and apply their own metrics as appropriate. We would be interested to correspond with anyone who does so.

1. Introduction

Process modelling is a potentially powerful technology which may be utilised in order to further understand, experiment with or control the development process. It has been suggested (Pfleeger, 1994) that process technology may be enhanced by combining process-modelling with software measurement. We can split the research into studies which have attempted to use the process model as a framework for collecting product metrics and those which use it as a framework for collecting process metrics (Shepperd 1992), (Shepperd, 1992b), (Lott, 1993c).

Recent work on combining process models and process metrics has utilised the modelling technique as a way of displaying the process measurement such that data and process form one coherent graphical model (Phalp, 1995). All of these techniques use the process terminology, phase activities, or boundaries to structure their data collection.

However, there has been relatively little work reported which uses these complementary disciplines to study software maintenance. This paper reports on a study which collected maintenance data with respect to four independent categories. (Perry et. al. (1994) have taken a similar approach although theirs was an examination of the process at a much more detailed level, and did not attempt to fit the information into a more generic framework as we have done.) The following

section describes the maintenance process studied and the rationale for its adoption. Sections 3 and 4 describe related work in this area and the application respectively. Section 5 outlines the method used to implement and measure the maintenance process and section 6 explains our data collection procedures followed by the results and analysis.

2. Incremental Prototyping

The work reported here is based on the *incremental prototyping* lifecycle. A prototyping phase defines the changes (required by the customer) to be made to the existing system, so that most of the existing system is left unmodified. Following the initial analysis, project development involves rapid design of a prototype which is then built and used. (It should be noted that the developers only produced prototypes for the critical parts of the system.) The repeated refinement of the prototypes corresponds to the views of Allman and Stonebraker (1982) who suggest that it is crucial for development to be directed towards achievable short-term goals. Although long-term targets are not always fully understood, such targets can be decomposed into manageable short-term goals which can provide the developers with morale-boosting progress.

Once prototyping had finished, the coding of the actual implementation commenced. A separate testing phase was not included. Instead, testing is incorporated into the coding, with the testing of the implementation including the same tests as those used with the prototypes. During development, the compilers were used to perform as much code-checking and debugging as possible and to produce a set of run-time tests.

One feature of this project is that the developers were encouraged to experiment with some of their own ideas as well as those of the customer. Therefore, during development there was some experimentation in the use of features of both the prototyping language and the implementation language.

According to Parnas (1979), *software engineers have not been trained to design for change*. As the second release was being evolved by the same developers who designed the original system, how does Parnas' statement relate to our work? We are of the view that although developers are trained to design for change, there is still difficulty in designing for change. Basili and Turner (1975) are of the opinion that even if the developers have undertaken a similar project, they will still find it difficult to produce a good design for a new system on the first try. They suggest that this problem may be practically approached by initially implementing a subset of the problem and then iteratively enhancing the implementation until the full problem has been delivered. Our project uses a similar idea except that prototypes are only iteratively produced for critical parts of the system. An additional consideration is that the use of the same team for this successive implementation results in an increase in the skill levels and the resultant increase in productivity.

Constraints in process support did not force reuse from the previous implementation. A subjective assessment based on functions and classes indicates that the developers only reused approximately half of the old code due to the temptation to produce new code in order to experiment with new methods. This approach to code-reuse matches the views of Allman and Stonebraker (1982) whose philosophy is that it is never too late to discard all existing code and start again.

3. Related Work

There are several arguments cited in the literature suggesting that the concept of the lifecycle is unsuitable for the development of evolving systems today. In the 1970s, the lifecycle concept of performing activities systematically supported the idea of careful planning prior to machine access in order to make effective use of the then-expensive computer resources (Agresti, 1986).

There are several other dated assumptions built into the conventional waterfall lifecycle including prototyping and granularity. The sequential view does not fully account for important process attributes such as iterations and feedback loops as reported in Curtis (1981) and Curtis (1987). The concept of the conventional software lifecycle has been significantly altered following the acknowledgement of the need for prototyping (McCracken, 1982), (Gladden, 1982). Agresti (1986) challenges the assumption that development follows a rigid sequence of activities from requirements specification through to coding and testing. Lifecycle models offer a large-grained view of the development process, and as such, whilst suitable as an overview, cannot represent critical lower-level details of a project. Curtis (1992) states that many smaller processes are overlooked when a lifecycle description is used. Processes may be examined in terms of a whole phase instead of the multitude numbers of sub processes used during the phase, giving a less detailed view.

Real software development processes often do not consist of phases which are distinct. Swartout and Balzer (1982) argue that the software methodologies which separate specification from implementation are unrealistic. They claim that *every specification is an implementation of some other higher-level specification*. The partitioning of the development process into phases such as specification and implementation is entirely arbitrary.

Due to these problems many of the ideas of the conventional software lifecycle model have been challenged and consequently largely rejected. However, much of the terminology introduced still remains and indeed we have applied such terminology to this project despite having rejected the application of the conventional lifecycle to our work.

The problem of relating software measurement to process modelling has also been addressed. Rombach (1990) states that models and measures are inseparable and planned improvement of quality requires measurably improved development processes. Lott (1993a) describes several software engineering environments which

use process models and measurement data to improve the manageability. The MVP project (Lott, 1993) is an example of a proposed solution to the problem of using a software process model to improve quality. The project involved developing a prototype process representation language to specify software processes.

4. The Application

The application used for this study revolved around an Electronic Point of Sale system developed with process support (Greenwood, 1996). The work began three years ago, and during this time approximately 5K NCSL were developed. Release 3 involving a new set of customer requirements is now under development. The data reported on here relates to Release 2 of the software which was developed two years ago. Thus the maintenance process involves dealing with a large amount of legacy code, maintaining and altering it according to customer requirements.

5. The Design Method

The software is modelled using a process algebra (Henderson, 1995) and an OO modelling language (Henderson, 1993) before translation into C++. We used ProcessWise Integrator (Bruynooghe, 1992) to develop a model of the evolution process. This PWI model (Greenwood, 1995) has two main objectives:

- Recording the effort involved in a modification;
- Ensuring the consistency of the component relationships.

Associated with this model are several roles played by the project team, two of which are now briefly described:

- *The Developer's Role* covers three types of actions. *Effort Actions* are measured by the time spent by a developer working on a particular component. The information required is an identifying name for the specific component and the type of effort action. (Effort was being recorded in 1/4 day units which corresponds to 1.5 hours excluding time for breaks.) *Agrees Actions* allow the user to record that two components in a relationship agree with each other. (Details of these actions are not relevant to the effort analysis contained in this paper.) *Change Actions* are used to correct any data which has been supplied to the model. The user is able to indicate that a component has been changed without recording an associated 1/4 day effort. All effort information was recorded by the developer after the effort had been expended. In future, PWI will automatically record effort information on-line.
- *The Measurer's Role* is the name given to the metrics team responsible for extracting the effort information of the developers from PWI's log. The role only has two possible actions. *Modify This Role* is a default action included by PWI which allows the possibility of making further changes to this role. *Output Effort Log* causes the effort log information to be extracted and written to a standard text file.

6. Data Collection

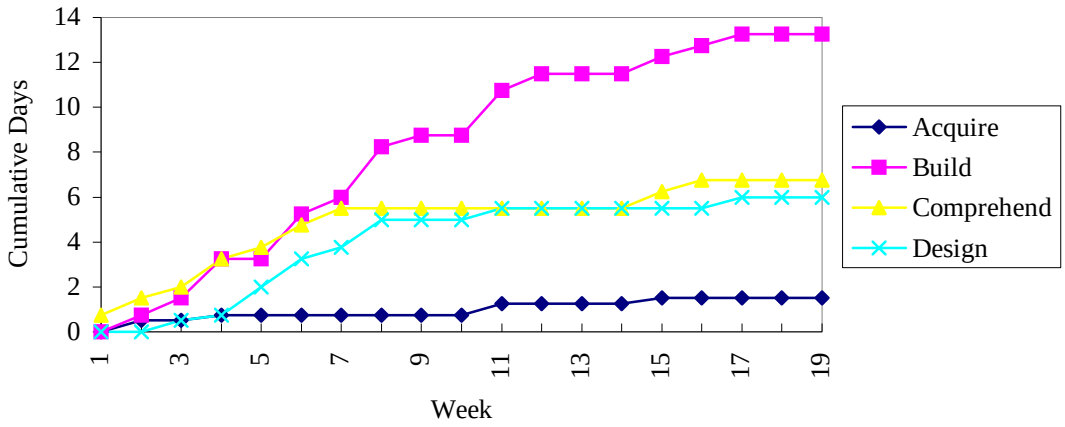
The actions of the developer are categorised for recording purposes into the following four types of effort based on the activities associated with the *Pumping Model* (Henderson and Warboys, 1991):

- *Acquire*: the acquiring and customising of existing software, including the acquisition of other people's code and coding techniques from the literature;
- *Build*: the coding of low-level modules, unit testing and the building of test harnesses;
- *Comprehend*: the understanding of the system, possibly involving literature surveys and experimentation with hardware and software;
- *Design*: the high-level design of models or platform software prior to coding, as well as integration-test planning.

7. Analysis and Results

7.1 Chronological analysis of developer activities

Figure 1



The graph illustrating the cumulative time spent by the developer per task across the component database (figure 1) shows that in the initial stages of this project, the most time-consuming activity was that of comprehension. Indeed, the developers' time was concentrated wholly on this activity of understanding during the first week of the project. Intuitively, this is the expected result, with the developers interspersing periods of time doing background reading with periods of experiment. During the first few weeks of the project (weeks 2 to 4), figure 1 illustrates that the developer also spent a small amount of time acquiring and customising existing software and tools to be used at a later date or experimenting with coding techniques described in the literature.

Figure 1 also shows that the relationship between the cumulative amount of time spent designing and the week number is almost linear between weeks 2 and 8. As

suggested by Curtis et al. (1990), part of this time spent on designing can be accounted for by team meetings. In our case, the developers met to exchange information and to discuss the details of the shared process support.

However, in the following period of the project (from week 9 onwards), the design was deemed to be finalised (apart from small amounts of time spent adding extra functionality to the design in weeks 11 and 17) and the developer's efforts were almost solely concentrated on coding. After a while, more of the available time needs to be spent on anti-regressive activities such as the restructuring of code and the updating of documentation. This may account for the time spent updating the design in weeks 11 and 17.

Gersick (1988) suggested that halfway through a group project, there is a critical point where the team comes to a consensus in order to make progress. This may be reflected by the delivery of the design and the concentration of the effort on coding. Flatter sections of the graph indicate periods of consolidation in the project. Once the task had been fully understood (around week 7), only a couple of occasional periods of time needed to be spent understanding the project in weeks 15 and 16.

Figure 1 also shows that the relationship between the cumulative time spent for the developer activity *Build* and time (project weeks) is almost linear throughout the duration of the project.

7.2 Chronological analysis of the project database

Figure 2

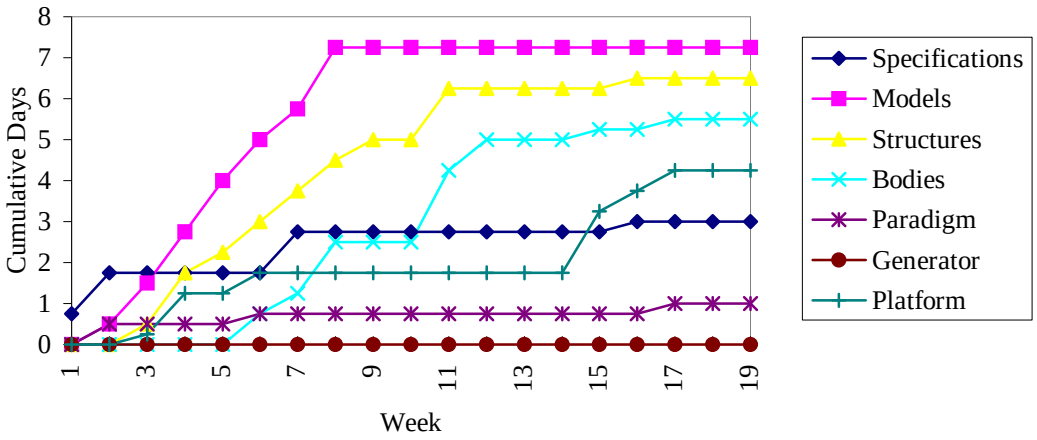


Figure 2 shows the plots of the cumulative time spent for each of the types of component in the database. The largest proportion of developer time was *Models*. Although for the early stages of the project, the relationship between the cumulative time for modelling and the week number is fairly linear, from week 9 no further modelling time was spent (corresponding to the critical point) and thus all subsequent

effort was concentrated on coding. This effort is captured by the *Code Structures* and *Code Bodies* components, which have accounted for the second and third largest proportions of developer time. Although no coding was done during the first week of the project, from week 2 onwards the relationship between the cumulative time spent on code interfaces and the project week number is fairly linear. At this time, the developers worked on the overall structure of the C++ classes to be used to implement the models. The production of code for the implementation of these classes *(Code Bodies*) was initiated much later (week 6).

Figure 2 also illustrates that the majority of initial project effort was applied to the specification, which was worked on in two periods of activity (weeks 1-2 and week 7). Again, from the critical point of week 8 onwards, the specification was finalised and all developer effort was channelled towards the implementation.

The components *Platform* and *Paradigm* only account for small periods of the developers' time early on in the project. Both of these were associated with the acquiring and comprehending the workings of existing tools and code. Figure 2 shows that no time at all was spent for the *Generator* components because the developers did not construct any tools for aiding the conversion of the model into code.

8. Conclusions

Following other recent work which has combined process models and process metrics, we have collected process data for each category of the process model. However, our decision to collect effort data against our four independent categories (*Acquire, Build, Comprehend and Design*) represents a departure from this orthodoxy.

Generally, the developer activity throughout this project followed the expected trends. Initially a large amount of the developer's time was devoted to tasks relating to comprehending the legacy products from previous releases. This leads us to suggest that project effort could be reduced by supplying additional documentation giving more details of the software's functionality.

All of this abstract activity information provides a distinct and orthogonal view of the developer's personal process. For example, we can see how comprehending the existing system spans a number of process activities or phases. This approach provides us with a much more detailed picture of the process than collection of data solely against the process model categories, and is also independent of the underlying process model. Hence it is invariant of process change and would be applicable to other situations irrespective of the project’s process.

Although our study is on a relatively small scale, our preliminary findings suggest that such an approach gives us a far greater understanding of the software development process than traditional approaches which only have an activity-based

view. Further work is continuing with both different developers and large-scale projects so that we can learn more about the nature of industrial software development and the applicability of our method.

Acknowledgements

The authors would like to thank Dr. Greenwood at the University of Manchester and Professor Henderson at the University of Southampton for their assistance throughout this project, and the EPSRC for their financial support.

9. References

Agresti, W. W., 1986, The conventional software life-cycle: Its evolution and assumptions, *IEEE Computer Society Press*.
Allman, E. and Stonebraker, M., 1982, Observations of the evolution of a software system, *IEEE Computer*, 27-32.
Basili, V. R. and Turner, A. J., 1975, Iterative enhancement: A practical technique for software development, *IEEE Transactions on Software Engineering*, 390-396.
Bruynooghe, B., Hook, P., Cook, P., Greenwood, R. M., Butler, P. and Parker, J., 1992 Processwise Integrator: Sun hosted system - 1.1, *ICL*.
Curtis, B., Kellner, M. I., and Over, J., 1992, Process modeling, *Communications of the ACM*, **35(9):** 75-90.
Curtis, B., Walz, D., and Elam, J., 1990, Studying the process of software design teams, *5th Software Process Workshop, Kennebunkport, Maine, USA, IEEE Computer Society Press*: 52-53.
Curtis, B., Krasner, H., Shen, V., and Iscoe, N., 1987, Substantiating programmer variability, *Proceedings of the Ninth International Conference on Software Engineering*, 96-103.
Curtis, B., 1981, Substantiating programmer variability, *Proceedings of the IEEE*, **69**: 846.
Gersick, C. J. D., 1988, Time and transition in work teams: Toward a new model of work development, *Academy of Management Journal*, **31(1):**9-41.
Gladden, G. R., 1982, Stop the life-cycle, i want to get off, *ACM SIGSOFT Software Engineering Notes*, **7(2):** 35-39.
Greenwood, R. M. and Warboys, B. C., 1996, Co-operating evolving components a rigorous approach to evolving large software systems, *Proceedings of the Eighteenth International Conference on Software Engineering.*
Greenwood, R. M., 1995 EPOS evolution process processwise integrator, *tech. rep., Department of Computer Science, University of Manchester, UK.*
Henderson, P. and Smith, A. P., 1995, Rapid prototyping of distributed systems, *tech. rep., University of Southampton*.
Henderson, P., 1993, Object-Oriented Specification and Design with C++, *McGraw-Hill.*
Henderson, P. and Warboys, B., 1991, Configuration description for component reuse, *1st International Workshop on Software Reuse, Dortmund, Germany*.

Lott, M. C., 1993a, Process and measurement support in sees, *tech. rep., Universitat Kaiserslautern*.
Lott, M. C., Pantelis, M., and Rombach, H. D., 1993, A mvp-l solution for the software-process modeling problem, *tech. rep., University of Maryland.*
Lott, M. C. and Rombach, H. D., 1993c, Measurement-based guidance of software projects using explicit project plans, *Information and Software Technology*.
McCracken, D. D. and Jackson, M. A., 1982, Life-cycle concept considered harmful, *ACM SIGSOFT Software Engineering Notes*, **7(2):** 29-32.
Parnas, D. L., 1979, Designing software for ease of extension and contraction, *IEEE Transactions on Software Engineering*, **5(2)**.
Perry, D. E. and Staudenmayer, N. A., 1994, People, organizations, and process improvement, *IEEE Software.*
Phalp, K. T., 1995, An investigation of process modelling in practice, *Ph.D Thesis. Bournemouth University, UK.*
Pfleeger, S. L. and Rombach, H. D., 1994, Measurement based process improvement, *IEEE Software*.
Rombach, H. D., 1990, Design measurement some lessons learned, *IEEE Software*.
Shepperd, M. J., 1992, Products, processes and metrics, *Information and Software Technology*, **34(10):** 674.
Shepperd, M. J., 1992b, Quantitative approaches to process modelling, Colloq. on Process Planning and Modelling, London, IEE.
Swartout, W. and Balzer, R., 1982, On the inevitable intertwining of specification and implementation, *Communications of the ACM*, **25**: 438-440.