An Empirical Study of the Evolution of a Software System

R. Mark Greenwood, Brian Warboys
Department of Computer Science
University of Manchester
Manchester M13 9PL UK
+44 (0) 161 275 6182
{markg,brian}@cs.man.ac.uk

Rachel Harrison, Peter Henderson
Dept. of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ UK
+44 (0) 1703 593249
{rh,ph}@ecs.soton.ac.uk

ABSTRACT

The cost effective and reliable evolution of systems is a significant software engineering challenge. Our approach is based on a combination of product modelling, process modelling and software metrics. We describe an empirical laboratory study following the evolution of three releases of a publicly available exemplar system. Analysis of the metrics which were collected improves our understanding of how a system evolves. Process metrics can provide information about how the product is put together, and product metrics suggest explanations for the development processes observed.

Keywords

Process Modelling, Software Metrics, Empirical Software Engineering, Maintenance, Evolution, Reuse

INTRODUCTION

Evolving software systems is a critical business issue. Businesses need to evolve their systems to respond to new opportunities and regulations. Cost effective and reliable evolution of such systems is a significant software engineering challenge. Many modern software systems are developed over a long period of time and, indeed if they are successful, their use provokes their own further development. Successful software systems are also rarely developed from scratch, but must incorporate existing components to be economically viable. Our belief is that cost effective and reliable evolution requires careful management of both the development and maintenance process, *and* of the product architecture.

In [3] we introduced a framework for evolving software systems based on an integration of product and process modelling. Each product component had its own evolution process, and the product design hierarchy was used to mediate the cooperation of these component processes. In this paper we describe a variation on this theme which explicitly addresses the fact that not all components evolve at the same rate, and evolving some components has a greater overall impact. To evaluate our approach we have applied it to a laboratory demonstrator, and gathered both process and product metrics of our activity.

The process metrics gave us information about the way in which the developers worked, allowing us to investigate differences between the idealised process model and the actual development process. This enables us to evolve the process in the light of how it works in practice. The product metrics help us to analyse the evolution of the system through three releases and to highlight anomalies. These provide future developers with an insight into the system's architecture which will facilitate evolution and maintenance. The combination of process and product metrics enable developers to identify the emergence of stable components as components become tried, trusted and *infrequently* changed. We believe that this will enable reuse effort to be targetted where it will be most effective. A good understanding of product structure is needed to know when to initiate an automated process in response to a human change and vice versa. The product and process metrics are analagous to factory sensors.

LABORATORY DEMONSTRATOR

We have used a laboratory demonstrator of an Electronic Retail System as the basis for our empirical study. We refer to the demonstrator as ERS. It is a distributed system covering both in-store and back office retailing functions. The first release, ERSr1, was developed from scratch. The second release, ERSr2, incorporated new requirements which emerged during the testing and demonstrating of ERSr1. The evolution from ERSr2 to the latest release ERSr3 followed the same pattern.

A disadvantage of using a laboratory demonstrator is that ERS is not itself a typical successful business system, so further work is required to check that our results scale up. The advantage is that we have control over the metrics collection, and can make the demonstrator publicly available [1] for comparative studies. We have attempted to mirror the conditions of the commercial system development in our empirical study. For example, meeting the deadlines for product releases had precedence over following the defined process.

A HIERARCHIC PRODUCT VIEW

ERS was designed by decomposing it into components, and by using a hierarchy to view the system at multiple levels of abstraction. The hierarchy for ERSr1 is shown in figure 1. We refer to this as an ERS *product tower*, and we believe that the evolution process for ERS should complement the understanding in the product tower [3].

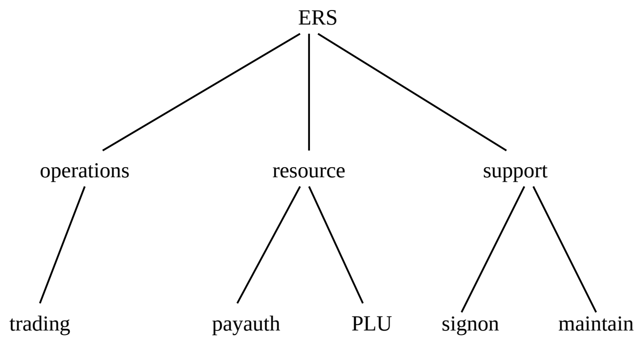

Figure 1: **A Product Tower for ERS**

Often one tower is not enough. For example, ERS uses a UNIX sockets package. This might be a separate system which is used in other products as well as ERS. From the perspective of ERS the sockets package is an existing system whose facilities it exploits. We term such a component a *platform component* since it is part of the base on which ERS is built. Some platform components may be outside the control of the system developers. The sockets package makes use of UNIX system calls, so UNIX is a platform component for the sockets package. The developers responsible for the sockets package cannot control the development of UNIX; rather they have to react to it. In the product hierarchy the emphasis is on decomposing a large complex system into components. Platform components add an emphasis on layering, each platform provides a set of abstractions for other components to exploit. These are complementary ways of dealing with the complexity of a large system. Whether a component is a platform component or part of ERS is not pre-determined. Over time parts of ERS, say the signon server, might be recognised as a good safe component and adopted by other systems, becoming a separate product tower and a platform component to ERS. Conversely, it may be decided that the existing sockets package cannot satisfy the high performance demands of ERS so it will be dropped and a specialised ERS-socket component developed.

A GENERIC PROCESS MODEL

Given any product tower, such as ERS, or a component within a tower such as PLU, we expect to find three things: a description of what it should do, its current code, and the platform components which it builds on. In ERS we have termed these the *specification*, *code bodies*, and *platform* respectively, and refer to them as different *views*. This enables us to distinguish separating a system into its different views, from separating it into a set of constituent components. Each of these three views can be the starting point for system evolution.

The related issues of size and architecture give rise to another three views in our generic evolution model. The *model* enables developers to reason about the system without having to cope with the details of the code bodies. The model abstracts from the complete system to emphasise the core architectural decisions. In a distributed system these might be the way the system is divided into components and the way that these are connected. The choice of what is modelled is reflected in the *paradigm* which has a dual role. The paradigm is the language for expressing the model, and it is an abstraction over the run-time facilities provided by the platform. The *code framework* is essentially the manifestation of the model in the language of the code bodies. It gives the overall layout of the code bodies based on the model and references the appropriate platform facilities where the model used their abstract equivalents in the paradigm.

The addition of *model*, *paradigm* and *code framework* to the *specification*, *platform* and *code bodies* gives our full generic model, shown in figure 2. The relationships are those which reflect major considerations during evolutionary development.

This can be illustrated through the use of development scenarios.

- *Scenario A* – Specification is changed. Is the current model still sufficient? If so then only the code bodies need to be changed. If not then the model, and possibly the paradigm, must be revised, then the code framework and code bodies.
- *Scenario B* – Platform is changed. Is the current paradigm still appropriate? If so then change the code framework to match the new platform and then the code bodies. If not then the paradigm, model, code framework and code bodies need to be revised.

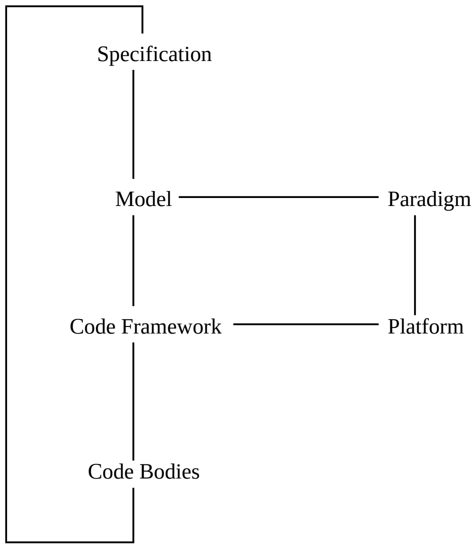

Figure 2: **Generic Model**

These scenarios are idealised. There are a number of reasons why development will not follow such a smooth course and clearly a development may involve more that one such scenario. The general pattern, however, is that changes introduce inconsistency between the different views of the system and the developers evolve the system to eliminate these inconsistencies.

ERS Specialisation of Generic Model

In developing ERS a number of specific decisions were taken. It was decided that the system should be coded in C++. During the first release it was decided to use UNIX sockets to connect the different programs within ERS so Socket emerged as a key part of ERS platform. The paradigm chosen was HOC [5], a derivative of the process algebra CSP. HOC supports the modelling of reactive systems in terms of processes, channels and events. The informal paradigm to platform mapping was that there would be a UNIX process for each HOC process, and the HOC channels would map onto a C++ Socket class which encapsulated the underlying system calls. One motivation behind the choice of HOC was that we had an implementation of HOC in Enact, an object-oriented modelling language [4]. This enabled us to execute the HOC models and so increase our confidence in the architecture of ERS.

DATA COLLECTION

The evolution of our ERS laboratory demonstrator was done by two researchers. One developed the initial ERSr1 and evolved this to ERSr2. The second took over ERSr2 and evolved it to ERSr3. In order to monitor this process, the developers kept track of how they spent their time. They recorded the *activity* involved, which *component* they were working on, and which *view* from the generic model was their focus. A process model was written and implemented using ICL's ProcessWise Integrator (PWI) system [2] to support this data collection. The PWI system allowed the developers to browse the data provided or output it in a standard format for analysis using other software. The system was used as a monitoring tool only. It did not attempt to prescribe the development process to be followed.

VALUE CHAIN MODEL

We have found the *value chain* [6] an effective way of depicting the systems' evolution. In contrast with the product tower, the value chain's primary focus is how different components' development processes interact. We use the term value chain to emphasise that every cell in the chain may be evolving its product concurrently, and periodically delivering its revised product to cells later in the chain. Value chains for the last two ERS releases are shown in figures 3 and 4. The new components added at each release are easily identified.

Annotating the value chains with metrics can help us to visualise the system's evolution. The figures in brackets below the names are: first the number of functions reused unchanged, and second the number of new or modified functions. In figure 3 (Release 2) the first figure in brackets is always low showing little reuse without change, whereas in figure 4 (Release 3) a significant section of components are unchanged (second figure is 0). This is indicated by the shaded area.

The value chains have also be annotated with the final developer's subjective assessments of each component's complexity. These are on a scale of C1 to C5, where C1 represents the simplest components and C5 the most complex. The main factor considered when producing the figures was *comprehensibility.*

The evolution from ERSr1 to ERSr2 included replacing the central PLU server with two PLU units to provide greater resilience. The two PLU servers were linked to ensure consistency. During this evolution the developer identified that the PLUs could be based on Stroustrup's associative arrays. A new component, Map, was created and the PLU component evolved to use this new platform facility. This and the decision to revise the Sockets component makes it inevitable that the changes will be wide ranging. In ERSr2 all the input from the cashier is handled by the trading program. In ERSr3 this is split into a "main" till program and three input programs, keyboard, scanner and balance. The till program

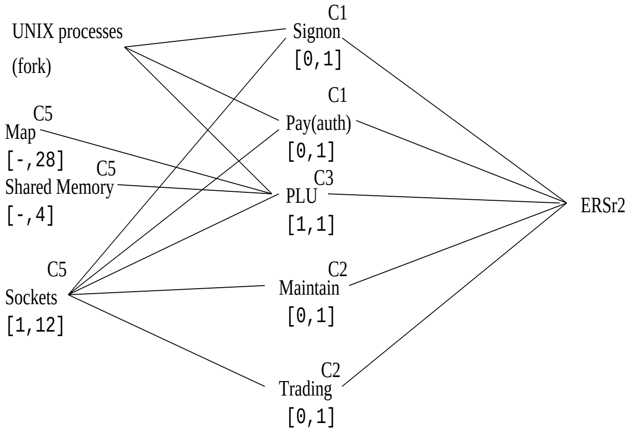

Figure 3: **Value Chain ERS release 2**

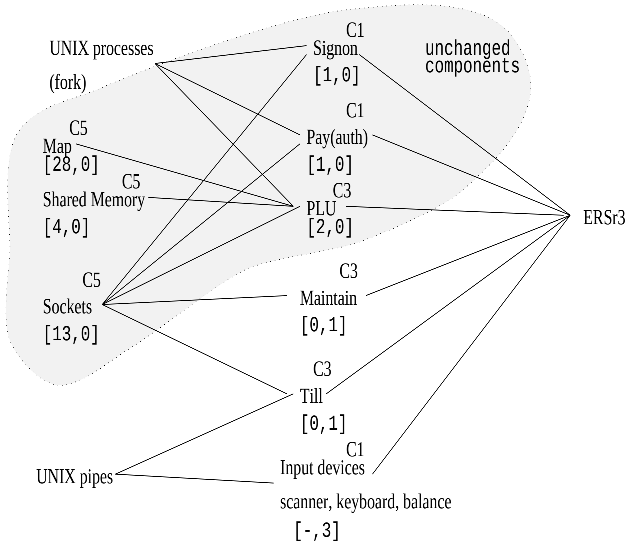

Figure 4: **Value Chain ERS release 3**

took over the socket connections of trading in ERSr2. It was connected to its input programs using UNIX pipes rather than sockets. Using pipes was a low cost solution: they were already available in the ERS run time environment, and their use simplified the input programs.

CONCLUSIONS

Our ability to evolve successful business systems is critical to the future of software engineering. We have performed a modest empirical study of the way in which software systems evolve. Our approach is based on a combination of metrics, process modelling and product modelling. Process and product models are used to define metrics which enable us to get a view of how the product architecture changes, and how developer effort is distributed. The metrics in turn indicate distinct areas for improvement which are addressed through product and process changes. Our component based approach to process and product models fits well with established techniques for coping with large systems. A key result was supporting evidence for our conjecture that integrated models of the process and the product architecture offer a better way forward than "standalone" software process models.

We invite other researchers to use our software [1] and its associated metrics for their own research, and would be delighted to correspond with those who do so.

ACKNOWLEDGEMENTS

This work was supported by UK EPSRC projects GR/J08928 and GR/J11034. The developers of the three ERS releases were Adrian Smith and Reuben Nithi of Southampton University.

REFERENCES

[1] ELSS project - Electronic Retail System demonstrator software, 1996. Available by anonymous ftp from <ftp.ecs.soton.ac.uk> in directory <pub/-peter/evolution>.

[2] R. F. Bruynooghe, R. M. Greenwood, I. Robertson, J. Sa, R. A. Snowdon, and B. C. Warboys. PADM: Towards a total process modelling system. In A. Finklestein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 293–334. Research Studies Press, 1994.

[3] R. M. Greenwood, B. C. Warboys, and J. Sa. Cooperating Evolving Components - a rigorous approach to evolving large software systems. In *Proceedings* 18^{th} *International Conference on Software Engineering*, pages 428–437, Berlin, Germany, March 1996. IEEE Computer Society Press.

[4] P. Henderson. *Object-Oriented Specification and Design with C++*. International Series in Software Engineering. McGraw-Hill, 1993.

[5] P. Henderson. Reconfigurable Dining Philosophers using Higher Order Communications. Technical report, University of Southampton, 1993. Available by anonymous ftp from <ftp.ecs.soton.ac.uk> in directory <pub/peter/various>.

[6] P. Henderson and B. C. Warboys. Configuration description for component reuse. In 1^{st} *International Workshop on Software Reuse*, Dortmund, Germany, June 1991.