Software Reengineering Patterns

Rob Pooley (rjp@dcs.ed.ac.uk) and Perdita Stevens (pxs@dcs.ed.ac.uk)

Department of Computer Science
University of Edinburgh
Kings Buildings
Edinburgh EH9 3JZ
Tel: +44 131 650 5123 Fax: +44 131 667 7209

Abstract. The problem of reengineering of legacy systems, in the widest sense, is widely recognised as one of the most significant challenges facing software engineers. So-called legacy systems are normally, but not necessarily, large systems built in an era before encapsulation and componentisation were regarded as fundamental tenets of design. Through a gradual process of accretion and change, they have become devoid of useful structure. This makes them hard, expensive or impossible to modify in order to meet changes in the business processes. Legacy systems, whilst often essential to the running of an organisation, also inhibit change in that organisation. The problems of legacy systems are not limited to any one kind of organisation: large corporations and SMEs both suffer. Moreover, there seems no reason to be confident that today's new systems are not also tomorrow's legacy systems. The problem of reengineering legacy systems is probably here to stay.
In this paper we introduce the idea of software reengineering patterns, which adapt the ideas of design patterns to identify lessons in successful reengineering projects and to make these lessons available to new projects. This is done in the context of component based reengineering, which has been the focus of considerable hope in the reengineering community, but which has delivered limited successes so far. These ideas are developed in terms of some introductory examples taken from real projects.

1 Introduction

Our aim is to understand the way in which experienced software practitioners undertake the component-based reengineering of legacy systems, so that we can develop better techniques and material for transferring expertise.

There is a great deal of expertise, in UK industry and elsewhere, but the nature of that expertise is too little understood, especially when it deals with systems whose structure must be incrementally improved, not abandoned. This is a problem, because the task of helping comparative novices to learn quickly to behave like experts relies on an understanding of expertise. In software design [5] etc., the term pattern has been imported from architecture to describe an application of an expert solution to a common problem in context. Learning the pattern includes understanding the context, the problem, the solution, and its merits and demerits relative to other solutions. Patterns have been adopted enthusiastically by software practitioners because a pattern is an effectively transferable unit of expertise. The vocabulary provided by patterns is also an aid to discussion and clear thought, by experts as well as novices.

The context of a reengineering pattern must of course be much broader than that of a design pattern, including business context as well as software context. We believe that with the help of industry experts it will be possible to identify such reengineering patterns, and that the beneficial effects of doing so will be both profound and wide-ranging.

Specifically, we aim:

- To identify a collection of important, validated reengineering patterns;
- To establish that the pattern paradigm is useful in reengineering.

We do not expect to develop a comprehensive pattern language immediately. However, we believe that if we can identify and disseminate the idea and some important patterns, then the urgency of the problem and the nature of the software engineering community will cause the work of identifying reengineering patterns to continue beyond our immediate project, with escalating benefits. The model of the adoption of design patterns shows that this is possible.

Thus, our work relates principally to the understanding, methods and strategies that experts already have for the problems of reengineering and these are what our patterns must describe.

1.1 Legacy systems

The problems that legacy systems pose in the UK and elsewhere are well known. Brodie and Stonebreaker[3] define a legacy system as one that significantly resists modification and evolution to meet new and constantly changing business requirements, regardless of the technology from which it is built. We wish to emphasise that not all legacy systems consist of millions of lines of COBOL.

The most widely researched and best-understood approach to reengineering legacy systems is “cold turkey” - the legacy system is replaced by a new system with the same or improved functionality. Indeed, many specialist companies make their livings from such work. Unfortunately, however, for a high proportion of large legacy systems such an approach is utterly infeasible. The risks of making such a huge change in a single step - including that business requirements inevitably change during the reengineering project itself - are daunting. Even more concretely, where a legacy system controls a large amount of mission critical data, the downtime that would be required for the cut-over, including the inevitable data scrubbing, may in itself be so unacceptable as to rule out cold turkey. Therefore, in many cases, an incremental approach may be essential.

1.2 Component-based reengineering (CBRE)

We have already implicitly distinguished reengineering from other kinds of modification. The main differences are:

- The reengineered system is supposed to be based on the engineering principles currently believed to be most sound. In software, this means, in our view, that it is component-based It is now almost universally accepted that a system that consists of a loosely coupled collection of highly cohesive components is easier to adapt than one that is not[1].
- The architecture and high-level design of the reengineered system are not identical with that of the original system: that is, the work cannot be regarded as routine maintenance.

[1] Unfortunately the best currently available definition of component seems to be “an easily replaceable part of a system” – that is, a component has the desired property by definition, if not by construction.

2 Application of CBRE to legacy systems

2.1 Motivation

The reason for undertaking reengineering is generally characterised as "business process change". Such change imposes new requirements on systems and imposes penalties in terms of inability to take advantage of new opportunities and to meet new obligations. We include in business process change, not only changes over time within one organisation, but also the situation - presenting many of the same problems - in which a system developed in one organisation is to be used in another.

Even when they work in organisations that, corporately, have a great deal of expertise and experience in evolutionary reengineering of systems to support business process change, software engineers have great difficulty in becoming expert. There is a shortage of books, papers and training courses that can effectively transfer applicable expertise (see section 3 below). Experts in reengineering are much rarer than are experts in design, and engineers in most SMEs will not have access to anyone with a significant amount of experience.

In software design, the major benefits of patterns are that they are small and specific enough for the community to validate them effectively, and also to function as useful learning units, whilst remaining abstract enough to apply in a variety of situations. We believe that the same benefits will accrue – and possible be even more important – from the identification of reengineering patterns. This view has been confirmed by our initial contacts with senior technical managers in industry. It is clear that there are patterns - i.e. situations which commonly arise and which are recognisable (consciously or unconsciously) to an expert and where the advantages and disadvantages of a particular solution are well understood by that expert. Furthermore, it is equally clear that some people are better able than others to talk in the somewhat abstract terms that describe such patterns. However, there seems to be no existing "pattern culture" for reengineering, so far as we have been able to find out, anywhere in industry.

2.2 A simple example

Let us begin with a simple concrete example: our initial contact with a major communications company suggested the following first draft of a potential reengineering pattern. This strategic, high-level example illustrates the conceptual difference between a design pattern and a reengineering pattern; other reengineering patterns could be more tactical, dealing for example with how to undertake componentisation in particular circumstances. This sort of reengineering pattern would be more closely akin to design patterns; indeed classification might sometimes be subjective. Further examples are given in Section 5 below.

Name: Divide and Modernise (might be revised on greater understanding)

Context: a legacy system whose technology (e.g. database) is obsolete and soon to be unsupported. An identifiable area of functionality, relatively well localised in the legacy system, of which a generalisation would be useful, but is not immediately mission critical.

Problem: Modification of a dying legacy system is undesirable. Wrapping the system, sometimes useful, is not a good solution here because it perpetuates the use of unsupported technology. If a new system is developed "from scratch" to replace part of the old, the developers will be expected to provide ideal functionality: it will be impossible to manage expectations and the project will become huge and correspondingly risky.

Solution: Begin by mechanically translating the relevant part of the database to a modern format. Rewrite the relevant code without yet attempting to change its structure, thus acquiring a new system providing part of the functionality of the old, but no more, and without substantially different structure from the part of the old system. Remove the now redundant data and code from the rest of the legacy system, handling the consistency and gateway issues. Then consider the reengineering of the now-separated, manageably sized system.

Consequences: Work proceeds in distinct manageable phases. Even if "requirements explosion" does overtake the final restructuring step, the main aim, that of removing the dependency of the functionality on the obsolete technology, will have been achieved.

Major outstanding questions include: in which cases can the problem of data dependencies between the new and the old system be solved, and how? Under what [business? technical?] circumstances is the restructuring of the new system [politically? technically?] possible and/or desirable?

3 Relationship to past and current research

The two main areas we have to consider are reengineering, particularly evolutionary reengineering, and patterns. To take the second first: even though patterns are a rather new import to software engineering, there is already a thriving patterns community, including a newly founded BCS Patterns Special Interest Group, and several active mailing lists. Some of the problems we shall face - in particular, that of finding the right level of abstraction at which to describe patterns - are generic pattern problems, and interaction with the pattern community will be invaluable.

In reengineering the situation is less healthy. There is a large amount of successful technical work on reverse engineering, and applications of this to "cold turkey" are straightforward. In recent years it has been widely accepted in principle that reengineering usually needs to be evolutionary, but to date, work in this area has been less successful. A symptom of this is that UK (and other) software engineers, including those in successful software organisations, still face great difficulties in learning effective techniques for evolutionary reengineering.

Brodie and Stonebreaker's useful book [3] discusses the problem in highly pragmatic detail, and proposes an 11-step "Chicken Little" methodology for evolutionary reengineering. It is characteristic of work in this area, though, that although this book proposes a methodology, its real usefulness lies in wisdom it imparts in the process of describing the methodology. This is unsurprising, since the contexts in which reengineering takes place vary so widely that it is difficult to imagine what a reengineering methodology, with a breadth of applicability and perceived successfulness comparable to present day development methodologies for new systems, would look like.

There are several projects worldwide currently seeking to develop such methodologies (including the ESPRIT RENAISSANCE project, which involves Lancaster University): but their task is extremely hard, and it is not clear whether they will ever bear fruit. We think that such projects would be well complemented by context-dependent techniques such as the identification of particular reengineering patterns (just as design patterns complement design methodologies).

Work which we consider would provide useful input to ours includes [3] makes a contribution to the problem of how to recognise whether evolutionary reengineering of a system is possible, and [2] which gives checklists of aspects of the environment which must be considered. Refactoring techniques discussed for example in [6] are relevant, though they have mostly been applied to comparatively small systems and it is not yet clear how far they can be extended. Most interestingly, recent work by O'Callaghan, at De Montfort University, in collaboration with BT, has considered the application of design patterns to migration to object technology: we expect there to be synergy with his work.

It seems clear that the time is ripe for identifying software reengineering patterns, but there seems to be no previous work on this specific topic The term "reengineering pattern" has already been coined by Michael Beedle in [1]. However, his work deals with patterns for BPR, not with the specific problems of systems reengineering – and is in any case somewhat preliminary.

4 Research methods used

It is vital that the patterns we identify and disseminate should be valid, in the sense that they really do describe expert solutions to problems that genuinely occur commonly. To ensure this, we need to make use of our collaborators' experts, and also of the wider software engineering community. We are using a variety of techniques:

1. Study of particular projects in industrial collaborators, using some or all of the tactics:

(a) Take part in and contribute to informal discussions of the project as it proceeds;
(b) Attend design reviews and other meetings of the project;
(c) Interview a senior designer on a project about the strategy they are adopting in the reengineering of a system, and why;
(d) Interview both senior decision-makers and junior engineers, at various stages of the project, about the progress of the project.

2. We expect the first two techniques to be the most useful, since they will not affect the progress of the projects adversely. Taking people away from their project work to be interviewed is unlikely to be practicable at the most interesting stages of the projects!
3. We will observe the problems that arise and the tactics that the project team use to address them, paying particular attention to any areas where the behaviour of the team seems to deviate from the strategy planned in advance.
4. Undertake a series of interviews of experienced designers, aiming to identify the patterns that they consciously or unconsciously use.
5. Solicit input and comments from the reengineering community and the patterns community at large, making appropriate use of workshops, conferences, mailing lists and newsgroups.

This process is as iterative as time and available projects permit. We try to draft candidate patterns based both on our observations and on what the experts tell us. We then test these candidates by observing whether they occur in the later projects we observe, and by discussing them with other designers. As software engineers ourselves, we are able to draw on our own experience, but this is not sufficient in itself to test candidates.

5 Further examples

The examples in this section relate a group who reengineered a set of existing programming language compilers, to produce a component based portable compiler suite. The work occupied a number of years and occurred in several stages. New requirements emerged as the demands of both processor manufacturers and compiler users changed.

This work benefited from the clear understanding of modularity in compilers, based on successive translation stages. This meant that, although no effort had been made in writing the existing systems to ensure low cohesion, it was comparatively easy to believe that this would prove possible to achieve.

The patterns proposed here are both closely related to possible design patterns which would emerge in constructing similar systems from scratch, since the reengineering pattern can be thought of as a reason for selecting a design pattern from among candidates.

5.1 A first example - Externalising an internal representation

Name: Externalising an internal representation

Context: A new, optional intermediate phase or set of phases is required in a system. The addition is to be made at the interface between two existing phases. Phases are entered under the control of a driver program, which is easily modified to invoke additional processing phases.

Problem: The system must still operate as before if the optional phase(s) is (are) not selected. Currently the system can use an internal representation to transfer data between the two phases. The new system requires this private format to be usable by phases that may be added in future.

Solution: Replace the internal format with an externally defined one, open to use by new phases. Record the output of the earlier of the existing phases in an easily readable medium, such as a file or a database. Ensure that any modifications performed by the new phases leave their output in the same format. Modify the existing phases to output and input the new format.

Consequences: The generation of an externally readable version of the representation allows new modules to be attached with no further alteration of the existing system. This may be slower than the original system when the new options are not selected, but creates a very open system.

5.2 A second example - Portability through backend abstraction

Name: Portability through backend abstraction

Context: A pair of stand-alone legacy systems whose current target (e.g. processor instruction set) is about to become a set of targets. Their main functionality, producing one of this set of targets from distinct inputs, is logically equivalent. It is anticipated that new targets will continue to emerge frequently.

Problem: Modification of a working legacy system is undesirable. Wrapping the system, sometimes useful, is not a good solution here because involves additional translations, from the old target to the new one, which are not easily defined and appear to be very error prone. If a new system is developed "from scratch" for each new target, the developers will be expected to meet demands from a rapidly evolving set of target projects.

Solution: Rewrite the current systems, preserving intact their existing front ends, but disentangling their backends. Ignore the low level formats, which are the ultimate targets and define an abstract target, suitable for both easy translation from the front ends and easy translation to the targets. At the same time produce translators from this abstract intermediate code to the currently urgent targets. Do not modify the current working system for the old target at this stage, but begin work, in parallel and at lower priority, on a backend translator for this also.

Consequences: At worst the reengineering of the existing systems is no worse than writing new systems from scratch for one new target. In practice it is likely to be less costly, since the abstract target is chosen in part to be easy to translate to. Work on the backend translators will involve some extra overhead in the case of a single new target, but will represent a significant gain for the second target. Future retargetting will be quicker, easier and more flexible. The quality of the backends will be higher since more resources will be freed to devote to this stage.

6 Conclusions

This paper has proposed reengineering patterns as a way of codifying and disseminating good practice in software reengineering. Based on some early candidate patterns, we believe that this approach is extremely promising. The important additional feature of a reengineering pattern is, perhaps, that it deals with the reason or reasons for selecting a strategy, rather than simply identifying a good design pattern. The best design for a reengineered system may not be achievable in practice, but a sensible compromise may be available.

There is clearly much to be done to develop this suggestion to be fully usable. In publishing these early candidates, we hope to stimulate discussion and criticism. The principal questions at this point are:

1. Is the patterns approach applicable to reengineering?
2. Are the patterns put forward here of the right sort or should we look at alternatives?
3. What other candidates are there?
4. How can we validate candidate patterns?

We hope that a debate around these issues can begin. Our intention is to start a collection of patterns on our reengineering patterns Website. You are invited to send comments and proposals to the authors and to consult
`http://www.dcs.ed.ac.uk/home/pxs/reengineering-patterns.html`.

References

1. Beedle, Michael "Pattern Based Reengineering", Object Magazine, 1997
2. Bergey, John K., Northrup, Linda M., and Smith, Dennis B. "Enterprise Framework for the Disciplined Evolution of Legacy Systems", Technical Report CMU/SEI-97-TR-007 (1997)
3. Brodie, Michael L., and Stonebraker, Michael "Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach", Morgan-Kaufman Publishers (1995)
4. Brown, Alan W., Morris, Ed J., and Tilley, Scott R. "Assessing the Evolvability of a Legacy System", CMU SEI draft white paper, 1996
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J. "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Professional Computing series, 1994
6. Opdyke, William Object-Oriented "Refactoring, Legacy Constraints and Reuse", presented at 8th Workshop on Institutionalizing Software Reuse (1996)

Links to on-line versions, where available, are at our Web site:
`http://www.dcs.ed.ac.uk/home/pxs/sweng.html`