

# 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  
The King's Buildings  
Mayfield Road  
Edinburgh EH9 3JZ

**Abstract.** The problem of reengineering of legacy systems, in the widest sense, is one of the most significant challenges facing software engineers. 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.

## 1 The problem

The problems that legacy systems pose in the UK and elsewhere are well known. We wish to emphasise that not all legacy systems consist of millions of lines of COBOL. Brodie and Stonebreaker[1] 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.

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.**

However, in practice organisations have great difficulty in making evolutionary reengineering of systems work (first time, every time). 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 reengineers.** There is a shortage of books, papers and training courses that can effectively transfer applicable expertise. 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 summary, we believe that the most important problem is not an absence of expertise from British and other industry, but the difficulty of **transferring that expertise to those who need it.**

## 2 The solution?

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 ([2] 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. Importantly, **patterns are small and specific enough for the community to validate them effectively.**

We believe that the same benefits will accrue – and possible be even more important – from the identification of *reengineering patterns.* 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. From our initial contacts with senior technical managers in industry, it is clear that there are indeed patterns in reengineering, that is, 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.

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. We think that the approach will at the very least be a valuable complement to the development of over-arching methodologies for reengineering.

### 2.1 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.

**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?

## References

1. Brodie, Michael L., and Stonebraker, Michael (1995) Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach Morgan-Kaufman Publishers
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley Professional Computing series, 1994