Systems Reengineering Patterns

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

Department of Computer Science
University of Edinburgh
Kings Buildings
Edinburgh EH9 3JZ

Abstract. The reengineering of legacy systems – by which we mean those that "significantly resist modification and evolution to meet new and constantly changing business requirements" – is widely recognised as one of the most significant challenges facing software engineers. The problem is widespread, affecting all kinds of organisations; serious, as failure to reengineer can hamper an organisation's attempts to remain competitive; and persistent, as there seems no reason to be confident that today's new systems are not also tomorrow's legacy systems.

This paper argues

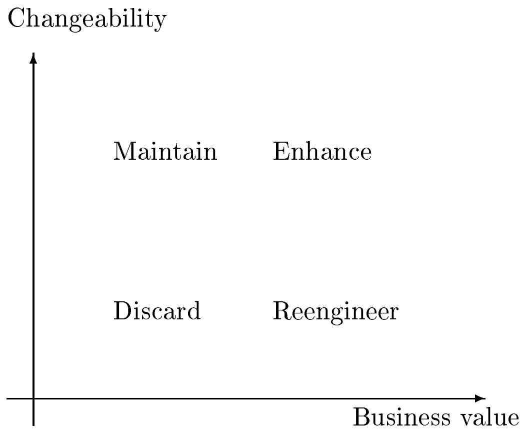
1. that the main problem is not that the necessary expertise does not exist, but rather, that it is hard for software engineers to become expert;
2. that the diversity of the problem domain poses problems for conventional methodological approaches;
3. that an approach via *systems reengineering patterns* can help.

We support our contention by means of some candidate patterns drawn from our own experience and published work on reengineering. We discuss the scope of the approach, how work in this area can proceed, and in particular how patterns may be identified and confirmed.

1 The problem

The problems that legacy systems pose are well known. Today's businesses can only survive if they can adapt rapidly to a changing environment and take advantage of new business opportunities. Since IT systems are now vital to almost all businesses, this requires that the IT systems can be modified quickly and cheaply. There are several reasons why this may not be the case. The best known is that the system was built in an era before evolvability was recognised as a difficult and crucial goal of system design. Many very large systems in COBOL, for example, fall into this category. However, even more modern systems which were originally well-built in an object oriented and/or component based way can become hard to modify, because the structure of the system becomes obscured by years of modifications. Brodie and Stonebreaker[4] 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, and this is the definition we will use.

What should be done with a legacy system? The now-classic decision matrix (see, for example, [10]) clarifies the options:



We are concerned here with systems which are good candidates for reengineering, because they are too valuable to the business to be discarded, but are too hard to change to be enhanced without restructuring. Of course, deciding which systems fall into this category is itself a skilled task, and one which has been addressed by [15], [5], [13].

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. This enables the reengineering problem to be factored into two phases: first, use reverse engineering and domain analysis to construct a new set of requirements, possibly identifying and retaining some aspects of the existing design such as the overall architecture; second, use an appropriate software development methodology to build a new system. Development is much better understood than reengineering, so the second step is comparatively tractable. Increasingly there is tool support available for the first. Unfortunately, however, for a high proportion of large legacy systems such an approach is utterly infeasible [4]. 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. Apprenticeship is probably the most effective way to learn; a software engineer is a member of a team for one reengineering project, gathering experience which will be helpful in running a latter project. However, 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. Recently the Y2K problem has exposed to many organisations their lack of corporate expertise in reengineering.

The problem of becoming expert is exacerbated by the wide range of factors (or *forces*) which must be taken into account in evaluating candidate solutions. Reliable statistics are scarce, but it seems that reengineering projects are even more liable than development projects to fail for reasons which can be considered "political": that is, they do not encounter any insurmountable technical problem, but rather, they fail to deliver sufficient apparent value to the business in a sufficiently timely way. A project may be cancelled even though it "would have" delivered a small enough cost/benefit ratio on completion, had it been allowed to run to completion. An expert reengineer is someone who understands how to prevent this, as well as how to deliver an appropriate technical solution.

In summary, we believe that the most important problem is not an absence of expertise but the difficulty of **transferring that expertise to those who need it**.

2 The solution space

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

Most work on systems reengineering so far has attempted to provide a *methodology* for reengineering. Examples include [4], [14], and Unisys' Refits. Each has example projects which have followed the methodology and succeeded. In most cases the projects have been undertaken in conjunction with the developers of the methodology as consultants or collaborators; comparatively little effort has been put, so far, into developing books or training courses that allow software engineers to learn to behave like experts. No reengineering methodology has yet had anything like the impact of the successful development methodologies. Reengineering methodologies are younger than development methodologies and it may be simply that not enough time has elapsed for adoption to occur. However, given that (especially because of Y2K) there is unprecedented industry attention to reengineering, it seems worth considering the other possibility, that we do not yet have an adequate solution to the problem. We can identify several possible factors:

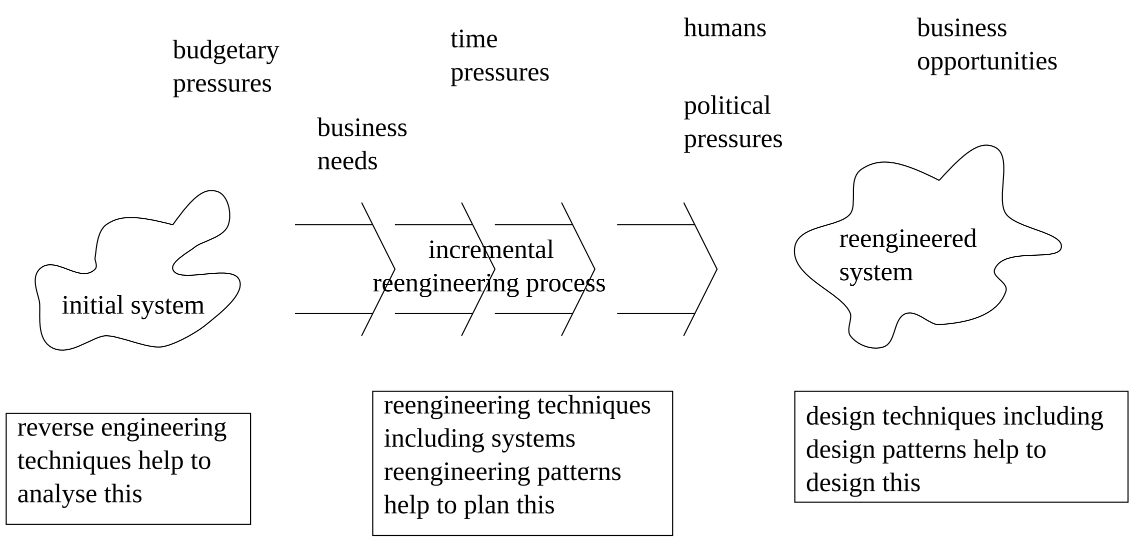
- Organisations and their projects differ very widely. One organisation's legacy system may be a small but vital and unmaintainable collection of spreadsheets, another's a system consisting of millions of lines of code. A methodology must either make (explicit or implicit) restrictions on its scope, or be huge, with most of the methodology being irrelevant to any given reengineering project. The potential user of the methodology then has to identify how to instantiate the methodology for their own project. In order to do this effectively an **understanding of expertise** embodied in the methodology is essential, but may be hard to obtain from published information about the methodology. In the worst case the assumptions may be unrecognised even by the authors of the methodology, since their own experience is drawn from projects in which their assumptions held.
- Effective validation of a reengineering methodology is very hard. The validation problem is harder than for development methodologies because of the greater dependence on social and political factors, and because of the large average size of the projects involved: people with experience in a very wide variety of reengineering projects are rarer than people with experience in a comparable variety of development projects.
- Software engineering research is particularly good at constructing (and evaluating) technical solutions to technical problems. Constructing and evaluating solutions to social and political problems is much harder: there is a tendency to evaluate contributions on the basis of their technical content alone. Of course general tools for handling social and political aspects are imported from other fields. For example cost benefit analysis is an essential part of planning a project. However, this can fail to take account of different stakeholders' viewpoints. Further techniques such as viewpoint analysis, and prioritising pieces of work described for example as use cases or scenarios, can sensibly be imported from software development. These techniques, however, are often under-emphasised in writings on reengineering.

In summary there is a great deal of expertise, 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**. This is particularly true where the learning has to be done by means other than apprenticeship, as in the case of an organisation which lacks significant reengineering expertise.

2.1 Proposal

In software design ([9] 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 *systems reengineering patterns*. By this we mean a description of an expert solution to a common systems reengineering problem, including its name, context, and advantages and disadvantages. In this paper we are principally concerned with the problem how *how reengineering should proceed*, rather than with what the design of the reengineered artefact should be. The latter problem seems to be adequately addressed by existing techniques (not least, design patterns). A reengineering pattern embodies expertise about how to guide a reengineering project to a successful conclusion. Because this is not only or even principally a technical problem, the context of a reengineering pattern must be much broader than that of a design pattern. It includes business context as well as software context, and may even need to take into account factors such as the budgeting procedures of the organisation or the personalities of the managers whose support is needed. 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.



We believe that with the help of 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. Patterns, being small and specific, may be validated individually, and information about the circumstances under which they are appropriate and their advantages and disadvantages, can be collated. An organisation, whose reengineering projects share a large number of characteristics, might make a collection of the patterns most useful to it, discarding any it considers inappropriate. Future methodologies might incorporate patterns which are appropriate to the assumptions underlying the methodology.

We do not expect to develop a comprehensive pattern language on our own – we are not experts in the whole field of reengineering. We think that academics like ourselves can have as their primary role that of *facilitating* the identification and validation of patterns by experts in the software engineering community. Even though patterns are a rather new import to software engineering, there is already a thriving patterns community, including conferences, local groups

and active mailing lists. We aim to harness similar energy to address the problem of reengineering legacy systems. This paper is one step in that direction; another is that we have set up a mailing list for discussion of systems reengineering patterns: see
`http://www.dcs.ed.ac.uk/home/pxs/reengineering-patterns.html`.

Patterns have also been adopted in several fields other than software design, some of which are relevant to systems reengineering. Cunningham's EPISODES [8] describes patterns for a process, in his case the software *development* process, emphasising the process of making decisions; an *episode* is a sequence of mental states leading to an important decision. Coplien has also worked in this area [7] and that of *organisational patterns* [12]. Appleton has written in [1] about patterns for *software process improvement*. In business process reengineering, the term *reengineering pattern* has been coined by Michael Beedle in [2] (which is why we have to use the slightly clumsy phrase *systems reengineering pattern* to describe our very different class of patterns!). So far as we have been able to find out this paper is the first to propose patterns as an approach to systems reengineering in the sense described here.

3 Scope and difficulties

Difficulties An important difficulty in identifying design patterns is that of finding the right level of abstraction at which to describe patterns. Experience in getting this right is growing in the design pattern community, and we try to learn from that experience here.

This work will share with all work in reengineering the difficulty of validating what has been done. It is easy to write guidelines – particularly the "motherhood and apple pie" variety! – much harder to find out whether they are correct and useful. We hope that the manageable size of patterns will ease this problem.

A further related problem, to which we have not found a solution, is that organisations are often unwilling to allow data about their reengineering projects to be published, especially when it relates to projects which were not completely successful.

What are systems reengineering patterns not?

Systems reengineering patterns are not design patterns Although design patterns are frequently useful to reengineering projects, the systems reengineering patterns we consider here are concerned with social and organisational issues as much as, if not more than, technical issues. Whereas a design pattern specifies something about the structure of the final[1] system, a reengineering pattern specifies something about the process by which the final system should be reached. Similarly, the applications of patterns to software development, to organisations, to process improvement and to business process reengineering cited above are interesting and relevant, but none addresses the particular combination of process, technical and organisational issues that arise in systems reengineering.

Systems reengineering patterns are not rules of thumb They should be supported by a discussion of their merits and demerits so that the reader can understand whether or not the use of a pattern is appropriate.

Systems reengineering patterns are not a methodology A systems reengineering pattern has a deliberately limited scope, and even a catalogue of reengineering patterns will not be a reengineering methodology, any more than a catalogue of design patterns is a design methodology. Eventually, experts will want to study both methodologies and patterns. Where it is best to start seems to be largely a matter of individual psychology, though lack of time may make a relevant pattern catalogue more attractive than a large methodology.

[1] a figure of speech: of course no successful system is ever really final!

Systems reengineering patterns are not formal objects Conceivably there could be cases where a technical description of a reengineering pattern, supported by rigorous argument as to why it was correct, could be useful. However, we do not yet have any example of such a pattern, and we think such things will be rare.

Systems reengineering patterns are not a panacea We propose them as a complement to, not a replacement for, other work in the area.

How can candidate patterns be identified? We propose the following techniques, which we have begun to use to identify our initial candidates, some of which are described in the next section:

- Study particular projects in industrial collaborators, using some or all of the tactics:
  1. Take part in and contribute to informal discussions of the project as it proceeds;
  2. Attend design reviews and other meetings of the project;
  3. Interview a senior designer on a project about the strategy they are adopting in the reengineering of a system, and why;
  4. Interview both senior decision-makers and junior engineers, at various stages of the project, about the progress of the project.

  We find the first two techniques the most useful, since they do not affect the progress of the projects adversely. Taking people away from their project work to be interviewed is unfortunately impracticable at the most interesting stages of the projects!
  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.
- Interview experienced reengineers about the projects they have been involved with, aiming to identify the patterns that they (consciously or unconsciously) use.
- Study published work on reengineering projects, extracting candidate patterns by abstraction from description of techniques that worked. Unfortunately such published work is in short supply.
- Draw on one's own experience of reengineering.
- Solicit input and comments from the reengineering community and the patterns community at large, making appropriate use of workshops, conferences, mailing lists and newsgroups.

How can patterns be validated? This requires collaboration with as many people as possible who have experience of reengineering. We can draw on our own software engineering experience as an initial "sanity check", but this is not sufficient in itself.

- Within our own research project, we can observe whether our candidate patterns occur in the later reengineering projects we observe. Since the number of projects that we will be able to observe directly in a small number of years is small, this technique is limited.
- Discuss candidate patterns other reengineers, in face to face interviews, on the mailing list and at workshops and conferences.

4 What kinds of things might be systems reengineering patterns?

In this section we propose four candidate reengineering patterns, drawn (in one case) from interviews with people in a large company which undertakes many reengineering projects, and (in three cases) from our own experience of working on reengineering projects. Comments, criticisms and suggestions from readers of this paper are welcomed, as part of the validation process.

Real validated systems reengineering patterns would be expected to be several times longer than these candidates. This is partly because we give abbreviated descriptions to fit the space available here. More importantly, the validation and elaboration process will identify more detail.

Patterns are described in a set format for ease of reference. At present several formats exist, differing in details. We use an abbreviated version of that used in [6], with elements:

Name: a few words, describing as evocatively as possible the overall nature of the pattern.

Context: a situation giving rise to a problem.

Problem: the recurring problem arising in that context.

Solution: a proven resolution of the problem.

Consequences: notes on the merits and demerits of the resolution described, with references to other possible solutions or relevant patterns where appropriate.

4.1 Divide and Modernise

This strategic, high-level example illustrates the conceptual difference between a design pattern and a reengineering pattern. It is drawn from discussion of several very large reengineering projects at the same commercial organisation, including verbal reports of the lively discussion which led one project team to the decision to follow the strategy described here, rather than developing a new system from scratch. We were fortunate to be able to talk both with someone involved with very high level strategic decisions about reengineering systems, and with people involved "on the ground" in one of the projects concerned.

Name: Divide and Modernise

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.
On the negative side, code and data is migrated before the new requirements on the system are analysed, which means that some of this effort may turn out to have been wasted.
Major outstanding questions include: can the problem of data dependencies between the new and the old system be solved, and how? Will the restructuring of the new system in fact be [politically? technically?] possible and/or desirable?

4.2 Modularity in compilers

The next candidate patterns are drawn from the experience of a group (including Pooley) 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.

In contrast to the previous example, the patterns proposed here are both closely related to possible design patterns which would emerge in constructing similar systems from scratch. These reengineering patterns can be seen as capturing the expertise embodied in the decision that it is possible and desirable to migrate in a certain incremental way to a new system which itself makes use of certain design patterns.

Name: Externalising an internal representation

Context: A system in which data is processed notionally in a number of phases, where the phases are invoked by a driver program which itself is easily modified. Phases are not currently well encapsulated, however: two phases which are currently always consecutive share an internal data representation which is adapted to the needs of those two specific phases, not designed to be an interface format for arbitrary processing. There is a requirement to add new (optional) phases which may intervene between two phases which previously were always called consecutively. It is expected that other new phases may be required in future.

Problem: Adding the new optional phases to the system as it stands requires either that functionality of the existing phases be duplicated, or that some optional phase use the "internal" format which was not designed as an interface format. Either course will create maintenance problems which are unacceptable in this context, given that there is an anticipated need to add further phases in future.

Solution: Incrementally replace the internal format with an newly defined and fully documented interface format, open to use by new phases. Modify the existing first phases to output the new format optionally. Develop the new optional phase using the new interface format as input. At this point the original first phase outputs two formats depending on what its successor will be. Next modify the old second phase to input the new format, at which point the old format can be abandoned, and the ability of the old first phase to output the old internal format can be removed. The structure of the new system is now modular, with the driver program as a Mediator[9].

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, apart from the easily modifiable driver program. This creates a more open system.
Depending on the nature and use of the old intermediate format, the new system may possibly be slower than the old, since the interface format is no longer so well adapted to the particular needs of the two originally communicating phases. If speed is critical, this effect needs to be considered in designing the new format and the altered phases.

The following example might be criticised for being too specific to a particular field – compilers – but that field seems broad enough to justify its inclusion here. Of course, an important benefit of the "piecemeal" patterns approach is that it supports the easy discarding of irrelevant patterns, those whose context does not apply to the reader.

Name: Portability through backend abstraction

Context: Compilers from two different languages to the same language, e.g. processor instruction set. It becomes desirable to support a new hardware platform, and it is anticipated that new targets will continue to emerge. Frequent minor modifications to the existing compilers are required.

Problem: 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: it is important to minimise the work needed to support a new platform and the maintenance work required. We could consider "wrapping" the old compilers and translating from the old target to the new, but such translations are not easily defined and appear to be very error prone. At the same time, it is impossible to cease to enhance the old compilers whilst developing the new ones.

Solution: Define an abstract intermediate target, suitable for both easy translation from the front ends and easy translation to the targets. Develop new versions of the two current systems, in which their frontends are preserved intact but they compile to the new abstract target. 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.

4.3 Changing interfaces in a client-friendly way

The final example draws on Stevens' experience of reengineering the Edinburgh Concurrency Workbench, which is a highly complex system which had evolved a structure which was clearly far from ideal, but where because of inadequate resources it was impractical to impose a new structure and newly designed interfaces in one go. It also embodies practice in APIs to large systems used in various versions by a number of developers: two examples familiar to us are Sun's Java Development Kit and emacs lisp. It seems to be less well known as a technique for use within a system.

Name: Deprecation

Context: Parts of a system are accessed using interfaces which are unsatisfactory: for example, the interfaces expose information which should be encapsulated, or they are inconsistent and hard to use. However, there is too much code using the interface to change the interface and all code using it in one go, or else the code which uses the interface is not under the control of the interface writer. It may not be possible to be completely confident that a particular modification to the interface is an improvement, until it has been tried out by a large group of users of the API.

Problem: The obvious solution is to modify the interface, release a new version, and force all clients of the interface to be modified accordingly. However, this may impose an unacceptable burden on the maintainers of those clients (whether or not they are the same people/organisation who own the interface). Worse, if a modification turns out to be a mistake – which may be hard to tell without full knowledge of how an interface is being used – it might be necessary to undo a modification, whereupon the double modification of the client code would be extremely wasteful of effort.

Solution: Using all available information, design a modification to the interface which is believed to be an improvement. Add any new elements to the interface. Any elements which are not present in the modified interface are not immediately removed, but are documented as "Deprecated" with pointers to alternative features which should be used instead. Users of the interface are encouraged to provide feedback on any problems they encountered using the new interface without deprecated features, particularly if this led client developers to continue using a deprecated interface element. The default procedure is that in each new release of the interface the features which were already deprecated in the previous release are removed; but feedback from users may provoke a rethink; for example, a feature which the interface designers had thought was not useful and had marked "deprecated" might turn out not to be redundant, in which its "deprecated" tag could be removed in a subsequent release.

Consequences: If cases emerge where it is difficult to avoid using a deprecated interface element, the element can be used and the reason for the difficulty examined. It may be that adequate replacement features are not in place. By deprecating the element rather than removing it we avoid presenting the API user with the frustrating situation in which a problem which was soluble using one version of the API becomes insoluble using a later version.

This technique is useful where the existing structure is reasonably sensible, but interfaces are poorly designed or too broad. It is harder to use it in cases where the structure needs to be redefined in a way which is visible to the user. In such a case facilities may have to be temporarily duplicated using the old and the new structure, which depending on the length of the deprecation period may be unacceptable.

Depending on the nature of the user community the deprecation may be ignored. One way to tackle this would be to specify that a deprecated interface element will be removed in a specific version.

5 Conclusions

This paper has proposed systems reengineering patterns as a way of codifying and disseminating good practice in systems reengineering. These patterns address the reengineering process, taking into account all the factors that affect the success or failure of a reengineering project, such as the urgency with which enhancements are needed and the priorities of the organisation. The intention is to address the problem in a way which takes into account the needs of a software engineer who needs to make decisions about reengineering in a reasoned way, taking advantage of the experience of others. The manageable size of patterns complements reengineering methodologies, in that an engineer can learn patterns individually as they become relevant, rather than having to learn a large methodology all at once. Systems reengineering patterns do not, however, remove the need for methodologies to embody other aspects of expertise, and we expect systems reengineering methodologies and systems reengineering patterns to influence one another. We have proposed some candidate patterns to illustrate the technique and its scope.

To take the idea further requires input from many sections of the reengineering community, and this paper solicits such input. You are invited to consult our new systems reengineering patterns Web page:
`http://www.dcs.ed.ac.uk/home/pxs/reengineering-patterns.html`
and to send comments or suggestions to the authors.

References

1. Appleton, Brad, "Patterns for conducting process improvement" In Proceedings of PLoP'97.
2. Beedle, Michael "Pattern Based Reengineering", Object Magazine, 1997
3. 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)
4. Brodie, Michael L., and Stonebraker, Michael "Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach", Morgan-Kaufman Publishers (1995)
5. Brown, Alan W., Morris, Ed J., and Tilley, Scott R. "Assessing the Evolvability of a Legacy System", CMU SEI draft white paper, 1996
6. Buschmann, Frank, Meunier, Regine, Rohnert, Hans, Sommerlad, Peter and Stal, Michael. "Pattern Oriented Software Architecture: A System of Patterns" Wiley, 1996.
7. Coplien, James O., "A Development Process Generative Pattern Language", in Proceedings of PLoP'95
8. Cunningham, Ward. "EPISODES: A Pattern Language of Competitive Development", available from `http://www.c2.com/ppr/titles.html`.
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J. "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Professional Computing series, 1994
10. Jacobson, Ivar, and Lindström, Fredrik "Re-engineering of old systems to an object-oriented architecture", OOPSLA'91.
11. Opdyke, William Object-Oriented "Refactoring, Legacy Constraints and Reuse", presented at 8th Workshop on Institutionalizing Software Reuse (1996)
12. OrganizationalPatterns web page, administered by Jim Coplien. `http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns`
13. Ransom, Jane, Sommerville, Ian and Warren, Ian "A Method for Assessing Legacy Systems for Evolution". In Proceedings of Reengineering Forum '98.
14. "The RENAISSANCE project" information and some documents available from `http://www.comp.lancs.ac.uk/computing/research/cseg/projects/renaissance/RenaissanceWeb`.
15. "Software Reengineering Assessment Handbook v3.0" available from `http://stsc.hill.af.mil/RENG/`

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