# Systems reengineering patterns

**Perdita Stevens and Rob Pooley, Dept of Computer Science, University of Edinburgh**

## The problem

The problems that legacy systems pose are well known, and we need not labour them to people involved in this workshop. Suffice it to say:

- Today's businesses can only survive if they can adapt fast, and this means their IT systems have to adapt too.
- Not all legacy systems are millions of lines of COBOL: even sanely built OO systems can and do become inflexible, for example if modifications have not preserved their architecture.
- The (often difficult) decision about whether to reengineer a legacy system (as opposed to replacing it with an OTS package, for example) has been addressed for example by [17], [5], [15].
- Technical and risk-management factors often make an iterative approach essential [4].
- In practice, **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. What there is tends to concentrate on the technical design aspects of the problem, underemphasising the process aspects, which have a strong human and political dimension.

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

Development projects fail as least as often for political as for technical reasons, and our impression (drawn from our own experience and that of our colleagues in the SEBPC programme) is that this is even more of a problem in reengineering projects. A challenge for identifying and codifying reengineering expertise is to capture both aspects of a successful solution, with the contexts in which it works.

Most work on systems reengineering so far has attempted to provide a *methodology* for reengineering. Examples include [4], [16], and Unisys' Refits. However, no reengineering methodology has yet had anything like the impact of the successful development methodologies. Possible problems (apart from immaturity of the field) are:

- **Organisations and their reengineering 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. Reengineering problems arise at many different levels, from the architecture of huge systems to the detailed structure of small components. 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.

## Patterns

In software design ([11] etc.) and more recently elsewhere, the term *pattern* has been imported from architecture to describe an application of a named 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**. It is important to understand that patterns are by definition not new. The purpose of a pattern is to organise validated expertise. Until a consensus is reached that a thing does embody expertise it is only a candidate.

We believe that if the academic and industrial reengineering communities pool their expertise to identify *systems reengineering patterns* the same benefits will accrue and possibly be even more important. This view is supported by the fact (which came to our attention very recently) that the Software Composition Group at Bern working in the FAMOOS project has come independently to the conclusion that reengineering patterns are worth investigating ([9] [10]); their focus is slightly different from ours, and our work should be complementary.

## What are systems reengineering patterns (not)?

**A systems reengineering pattern describes how to undertake the *process* of reengineering in a particular *business* and technical context.**

**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" 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 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. Even a catalogue of reengineering patterns will not be a reengineering methodology, any more than a catalogue of design patterns is a design methodology. Even pattern languages will not replace methodologies -- and we believe that credible pattern languages will require years of work and revision.

**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.

## 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. The wider patterns community well understands the importance of validation: **patterns are by definition not new**. The purpose of a pattern is to organise validated expertise, and until a consensus is reached that a thing does embody expertise it can only be a candidate. Level of abstraction is crucial. The twin dangers are writing ‘‘motherhood and apple pie’’ guidelines which are not useful, and writing a detailed technical description, derived from one or two examples, which almost never applies. Wider community involvement is essential to identify the factors that make a pattern abstract enough to be frequently applicable but precise enough to be helpful.

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.

## How can candidate patterns be identified?

We have proposed and begun to use various techniques, of which the most relevant here is:
- **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 with other reengineers, in face to face interviews, on the mailing list and at workshops and conferences.

We have set up a web page and a mailing list for discussion of systems reengineering patterns: see http://www.dcs.ed.ac.uk/home/pxs/reengineering-patterns.html.

## What kinds of things might be systems reengineering patterns?

In this section we propose two candidate reengineering patterns. We emphasise that they are candidates only. Real validated systems reengineering patterns would be expected to be 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].

These related examples draw (among other sources) 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. The restructured system consists of a set of utilities needed by many components (such as means of interacting with the user, registering commands, providing help) and some components. Some components are optional (for example, the one that calculates testing equivalences, which only some users are interested in); others exist in several versions (for example, the one that defines the process algebra being used: different users use different process algebras).

### Componentising by building a facade

This is a particularly common example of a technique long known to the *refactoring* community: impose a design pattern on an existing system. The problem which refactoring does not address, however, is how to achieve this gradually in an environment where it is impossible to impose the pattern in one go, because of resource shortage and a need for continual functionality enhancement.

**Name:** Componentisation by building a facade

**Context:** A "spaghetti object" system which should be componentised, for example, in order to be able to use alternative versions of part of the system, whilst continuing to enhance the system.

**Problem:** When it is unacceptable to stop enhancing a system for long enough to rearchitect it in one go, the rearchitecting tends to get put off. Meanwhile the problem gets worse.

**Solution:**

- Identify the responsibilities which the potential component should have.
- Identify all the functionality currently being provided by objects in the component.
- Add a Facade object which provides access to all that functionality, whether or not it logically belongs to the new component.
- Gradually, as time permits, remove operations from the Facade's interface that should not be there, in a sequence of small refactoring steps, modifying clients accordingly. In parallel, continue with essential modifications of the system, but avoid introducing any new undesirable operations into the interface. Make sure that every step is identified as either a refactoring step or an enhancement step.
- As the interface to the Facade shrinks, consider whether there is a need for a separate Facade object at all, or whether its role can better be played by an existing object in the system.

**Consequences:** There is always a running system which it is possible to enhance.

**Example:** The Agent subsystem of the CWB was given a temporary facade AgentWrapper. Later, after more than a year of gradually increasing the encapsulation, it became clear that there was no need for a separate Facade object; the Agent interface itself included all the operations required.

**Changing interfaces in a client-friendly way**

This example is not unique to object oriented systems, but it is particularly relevant to them.

**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.

**Example:** This is common 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 has also been useful on a smaller scale in the CWB.

# References


**1** Appleton, Brad, ''Patterns for conducting process improvement'' In Proceedings of PLoP'97, available from `http://www.enteract.com/~bradapp/docs/`

**2** Beedle, Michael ''Pattern Based Reengineering'', Object Magazine, 1997, available from `http://www.fti-consulting.com/users/beedlem/pbr.html`

**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), available from `http://www.sei.cmu.edu/reengineering/pubs/97-TR-007/`

**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, available from `http://www.sei.cmu.edu/reengineering/pubs/white-papers/BrMT96/`

**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, available from `http://www.bell-labs.com/user/cope/bibliography.html`

**8** Cunningham, Ward. ''EPISODES: A Pattern Language of Competitive Development'', available from `http://www.c2.com/ppr/titles.html`.

**9** Demeyer, S., Rieger, M., Tichelaar, S. ''Three reverse engineering patterns'', draft available from `http://www.iam.unibe.ch/~famoos/`

**10** Ducasse, S., Nebbe, R., Richner, T. ''Type-check elimination: two reengineering patterns'', draft available from `http://www.iam.unibe.ch/~famoos/`

**11** Gamma, E., Helm, R., Johnson, R., Vlissides, J. ''Design Patterns: Elements of Reusable Object-Oriented Software'', Addison-Wesley Professional Computing series, 1994

**12** Jacobson, Ivar, and Lindström, Fredrik ''Re-engineering of old systems to an object-oriented architecture'', OOPSLA'91.

**13** Opdyke, William Object-Oriented ''Refactoring, Legacy Constraints and Reuse'', presented at 8th Workshop on Institutionalizing Software Reuse (1996), available from `http://www.umcs.maine.edu/~ftp/wisr/wisr8/papers/opdyke/opdyke.html`

**14** OrganizationalPatterns web page, administered by Jim Coplien. `http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns`

**15** Ransom, Jane, Sommerville, Ian and Warren, Ian ''A Method for Assessing Legacy Systems for Evolution''. In Proceedings of Reengineering Forum '98.

**16** ''The RENAISSANCE project'' information and some documents available from `http://www.comp.lancs.ac.uk/computing/research/cseg/projects/renaissance/Rer`

**17** ''Software Reengineering Assessment Handbook v3.0'' available from `http://stsc.hill.af.mil/RENG/`


Remember to see also our Web site:

`http://www.dcs.ed.ac.uk/home/pxs/sweng.html`