Title: Identifying and communicating expertise in systems reengineering: a patterns approach

Rick Dewar*
Division of Informatics, University of Edinburgh
JCMB, King's Buildings
Mayfield Road
EDINBURGH EH9 3JZ
Scotland

Ashley D. Lloyd†
Management School, University of Edinburgh
50 George Square EDINBURGH EH8 9JV Scotland

Rob Pooley‡
Computing and Electrical Engineering, Heriot-Watt University
EDINBURGH EH14 4AS
Scotland

Perdita Stevens§
Division of Informatics, University of Edinburgh
JCMB, King's Buildings
Mayfield Road
EDINBURGH EH9 3JZ
Scotland

June 1, 1999

Abstract

The reengineering of legacy systems – by which we mean those that have value and yet "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. A complex mixture of business and technical factors must be taken into

*rgd@dcs.ed.ac.uk
†A.D.Lloyd@ed.ac.uk
‡rjp@cee.hw.ac.uk
§Perdita.Stevens@dcs.ed.ac.uk

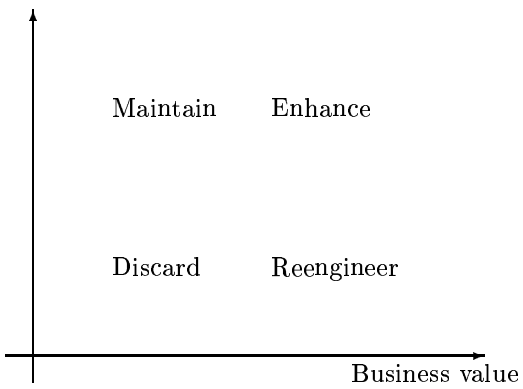
account to ensure success, and there is a wide range of different contexts each with its own problems. Moreover, the business needs do not stay constant whilst the technical factors are dealt with. In this paper we argue that the main problem is not that the necessary expertise does not exist, but rather, that it is hard for software engineers to become expert in all of the necessary areas. We propose that *systems reengineering patterns* may help to codify and disseminate expertise, and that this approach has some advantages over conventional methodological approaches. We support our contention by means of some candidate patterns drawn from our own experience and supported by information from elsewhere.

1 Introduction and background

In [1] the authors observe that despite an increasing level of activity in research in reengineering, "reengineering research has had notably little effect on actual software reengineering practice". In addition to the reasons that they identify – such as the difficulty of validating the research – we suggest that a major problem is that research results are difficult to communicate to the people who might make use of them. The wide scope of the reengineering problem and the absence of a commonly agreed classification of its areas exacerbate the problem. It is difficult for someone who wishes to make use of research in reengineering to find the material which is relevant to their particular problem.

Nevertheless there is a real problem to solve. Recently the Y2K problem has exposed to many organisations their lack of corporate expertise in reengineering. This is, however, a particularly simple example of a reengineering problem, as it is unusually technical in nature. More typically, the reengineering expert must consider system(s) which interact with the business's procedures in complex ways. This makes the systems reengineering problem more difficult in several respects. The now-classic decision matrix Figure 1 (see, for example, [2]) clarifies the choices that have to be made about what to do with a legacy system considered in isolation:

Figure 1: Decision matrix



but does not consider how the business value of a system may be changing, for example because of concurrent changes in the business processes it supports. Because we are concerned with systems reengineering, we are most interested in the class of 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. Even without considering changing business processes, deciding which systems fall into this category is itself a skilled task, which has been addressed by [3], [4], [5]. In general, the decision about which systems or parts of systems are good candidates for reengineering has to be

taken in the context of knowledge or expectations about how business processes are changing, and vice versa.

This complex interaction is particularly obvious when the organisation is undergoing *business process reengineering* (BPR); however, recent research has shown that there is quite a wide variation in the degree of importance placed on the existing processes when defining the new organisational system. Over 30 years ago Heany observed [6] that the process should be redesigned before the information technology support can be appropriately specified, since automating a mess produces, at best, a faster mess. Organisations still need to express what their processes are intended to achieve; however, there has been a change of emphasis in the final stages of system re-design. Increasing availability of COTS solutions has produced an economic incentive to fit the redesigned business process to the standard solution. This has had mixed success. Standard accounting solutions, perhaps benefiting from regulation compliance, have demonstrated clear efficiency gains. However, systems that deal with a company's core competitive competencies, such as production planning systems, may encounter a wider range of interfaces that are specific to the company. These require heavy modification of the software, which can ultimately eliminate the economic advantage. One manufacturer implemented a standard solution as a replacement to its existing systems and found that over 70% of the interfaces had to be specially programmed, a significant effort with no resulting increase in functionality [7].

Though this is certainly a cautionary tale, the same article [7] reports that, of the implementation costs of manufacturers who were identified as making the most effective and efficient use of IT, 75% were devoted to integrated standard software. Standard solutions are clearly effective, so the best approach may not be to identify the requirements of the business process and modify the software to suit: finding a good solution may require several kinds of flexibility. To be successful at providing a solution that is properly balanced between 'maintain', 'enhance', 'discard' and 'reengineer', the expert must understand how the existing system supports the current business processes, and what the effects will be of the various options for change.

The 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 [8]. The risks of making such a huge change in a single step – including that business requirements and/or processes 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.**

Reengineering, especially incremental reengineering in the presence of a changing business environment, is hard chiefly because of the wide range of factors (or *forces*) which must be taken into account in evaluating candidate solutions. An expert in reengineering is someone who understands how to deliver an appropriate technical solution whose business value is clear throughout the project: reengineering projects are particularly prone to 'political' failure. 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 later 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.

Earlier work in this project was reported in conference proceedings [9]. The present paper takes a wider, less purely software engineering view of the subject, and gives more detail, e.g. fuller versions of the patterns. Recently Bern's Software Composition Group has independently considered using patterns for reengineering [10]; their focus so far is on some specific technical issues in reengineering object oriented systems, however.

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 Patterns as an approach

We aim to understand how experienced practitioners undertake the reengineering of legacy systems, so that we can develop better techniques and material for transferring expertise. In particular, we want to address the problem of synthesising expertise in 'pure' systems reengineering with that which concentrates on the organisation, to help people acquire both in parallel. This has proved difficult in the past; both areas of study are large, and tend to be addressed by different communities. As we have explained, however, both are important to practical reengineering problems.

Given the recent phenomenal success of *design patterns* as a means of codifying and communicating expertise in design, it is natural to consider patterns as an approach also to the reengineering problem. Indeed, patterns have been adopted in several fields other than software design, some of which are relevant to systems reengineering. Cunningham's EPISODES [11] 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 [12] and that of *organisational patterns* [13]. Appleton has written in [14] about patterns for *software process improvement*. In business process reengineering, the term *reengineering pattern* has been coined by Michael Beedle in [15] (which is why we use the

slightly clumsy phrase *systems reengineering pattern* to describe our very different class of patterns). We think that patterns will complement 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. They can be domain-specific, helping practitioners to select relevant patterns. They may even be organisation-specific; and pattern-writing may contribute to the knowledge management process. Future methodologies might incorporate patterns which are appropriate to the assumptions underlying the methodology. An important advantage is that patterns, by codifying a manageable amount of expertise involving both business and technical factors and forces, may be helpful in the synthesis of work in these areas.

2.1 The scope of systems reengineering patterns

At present we prefer a broad interpretation of what might constitute a valid reengineering pattern. We propose, however, some ideas on what *does not* count as a systems reengineering pattern.

Systems reengineering patterns are not design patterns

Design patterns are frequently useful in reengineering projects, but 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 target system, a reengineering pattern specifies something about the process by which the target 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.

Having said this, there is an important class of reengineering problem involving decisions about whether and how to introduce a design pattern into a system, especially when the introduction has to be done in several stages.

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 is partly a matter of individual psychology, though lack of time may make a relevant pattern catalogue more attractive than a large methodology.

In the future we may hope to have a pattern language for reengineering; that is, a collection of patterns which work together in a synergistic way, because their interactions are well understood. (See section 3.1 below.)

Systems reengineering patterns are not formal objects

We do not mean to imply that there is no place for formality within a pattern description. Precise models of busines processes, of other aspects of the business context, and of the design of the systems involved will all at different times be useful.

However, we must remember what makes patterns useable: that they describe a solution – which may be described formally – in conjunction with an essentially informal discussion about what is good and bad about the solution in particular environments. (A particularly interesting example of rigorous design pattern work which does *not* fall into this pitfall is [16]: it would be interesting to see whether this kind of approach can be extended.)

The same danger is present in reengineering patterns. Many relevant facts about the business environment, in particular, are not likely to be formalisable in practice but must be included. In our current work we focus on careful, but informal, pattern descriptions.

Systems reengineering patterns are not a panacea

We propose them as a complement to, not a replacement for, other work in the area. We believe that they will add value to other approaches.

3 Developing patterns and pattern languages

An important difficulty in identifying design patterns is that of deciding what should be in the description and what should be abstracted away. Experience in getting this right is growing in the design pattern community, and we try to learn from that experience here. (The constructive criticism that we received on earlier versions has already helped us to improve the examples presented here: for example, our early examples were justifiably criticised for abstracting away so much that it was difficult to identify the cases where the pattern applied.)

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.

There are several areas of disagreement in the patterns community which have implications for our work too. The deepest concerns what constitutes a pattern: must a pattern embody knowledge which is widespread and uncontroversial in the relevant community? (This is the "patterns are by definition not new" approach which was our own starting point.) Or may a pattern embody a

new technique, an advance on the state of the art? Discussion of this question, prompted by the different initial approaches of our group and the Bern Software Composition Group, came to the conclusion that both approaches are valuable. It is important, however, that the reader of a pattern should understand how controversial it is: for example, while an expert might be most interested in novel patterns which presented new ideas, someone less experienced would be better served by trustworthy, uncontroversial patterns. We recommend that pattern descriptions should include a *Status* section describing the confidence which may be placed in the pattern.

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. The poses problems for the pattern writer, who wishes to include a *Known Uses* section in the pattern; this section describes cases where the pattern has been used successfully, giving specific names of products or companies. The need to include such a section is a good discipline, ensuring that patterns have some claim to applicability. However, finding examples where it is permissible to give enough detail to be helpful can be a problem; we do not have such an example for *Divide and Modernise*, for example.[1]

Some experts hold that a pattern should always include a motivating Example. We are not sure that this is always helpful; one of the examples given in full here includes one, the other does not.

Patterns are described in a set format for ease of reference. What that set format should be is a matter for debate. Process patterns, like [12, 14], tend to follow Alexander's original patterns in having a relatively unstructured format. They also tend to be short (typically under one page), because to make them longer tends to over-specialise the problem to the point of not being widely applicable. Design patterns on the other hand tend to have a highly prescribed format and to be longer, since they have to describe implementation techniques in some detail [17, 18]. Our experience so far suggests that it is useful to maintain the structured form of a design pattern, with modifications discussed above, and that reengineering patterns tend to be intermediate in length between process and design patterns.

Even for design patterns several formats exist, differing in details. We use a variation of that used in [17], with elements:

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

Status: a few words, describing how well established the pattern is.

Example: which may be made up for illustrative purposes.

Context: a situation giving rise to a problem.

[1] We have seen it several times, however, and believe it to be common: we would be grateful for citable examples!

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.

Known uses: of this pattern.

3.0.1 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.

  The first two techniques may be 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 often impossible at the most interesting stages of the projects!

- Using our observations gained above, we hope to 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. Linda Rising and Jim Coplien, quoted in a mailing list[2] discussion about how design patterns are identified, contributed some particularly useful questions that also seem relevant to reengineering patterns:

  - What would you share with someone you are mentoring?
  - What would be lost to the company if you left tomorrow?
  - What problems have you solved successfully on several projects?

[2] patterns-discussion@cs.uiuc.edu

- What have [you] done a thousand times that [you] think everyone knows?
- What do you say repeatedly at project meetings that never gets documented?

An additional point for reengineering patterns is that reengineering projects are often not identified as such by the people carrying them out: it is sometimes necessary to discuss modifications to systems more generally to elicit the examples.

- 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. A variant is to mine work on reengineering methodology for patterns.
- Draw on one's own experience of reengineering, preferably in consultation with people outside the particular project.
- Solicit input and comments from the reengineering community and the patterns community at large, making appropriate use of workshops, conferences, mailing lists and newsgroups.

3.0.2 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. Indeed, the whole point of patterns is not that they are amenable to formal validation, but that they allow the sharing of experience concerning success in a field of endeavour. Validation equates very closely with acceptance in the relevant community.

- 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 few 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.

3.1 Developing a pattern language

The ultimate aim is to develop a pattern language: that is, a collection of patterns which work together effectively in documented ways. The patterns are the words of such a language; the ways in which they can work together are the syntax and grammar. Thus a pattern language, to add value to its individual patterns, must be more than just a collection of patterns. Like programming language design, pattern language design involving non-trivial numbers of patterns is a hard problem, which moreover is insufficiently understood to date.

Some good (not formally published, but available) sources discussing the development of pattern languages are [19, 20].

What should a reengineering pattern language look like? This section is necessarily speculative: we describe some early ideas on the subject. There are several ways in which patterns may be related.

Two patterns may present alternative solutions to problems which are closely related, so that someone faced with such a problem would want to consider both solutions. We do not yet have examples of this from within reengineering, but the phenomenon arises in design patterns (consider Presentation-Abstraction-Control and Model-View-Controller [17], for example) and in more general development process patterns. For example we have recently begun to observe patterns WarRoom and WorkShop, which describe different possible approaches to solving severe communication problems between parties to a development, which are appropriate in subtly different circumstances.

Two patterns may be particularly useful when applied in conjunction with one another. For example, *Externalising an Internal Representation* (described below) and *Portability through Backend Abstraction* (briefly described in [9] – unfortunately space restrictions forbad including it here) are complementary.

More revealingly, patterns which deal with problems which occur at different levels in the organisation, as the concern of different individuals, may synergise. More work is needed here but as a preliminary example, one could imagine writing a business-level pattern which described circumstances in which a good strategy was to turn an existing product into something which could evolve more readily in response to user demand, and discussed the business-level changes required to do this. Such a pattern might refer to Deprecation (described below), which is a technical systems level reengineering pattern describing how to manage the updating of an API.

It is interesting to note that patterns which are not themselves reengineering patterns may be usefully connected to those which are. It may be that rather than thinking of a reengineering pattern language, it is more appropriate to develop reengineering patterns as part of a wider-ranging pattern language for software intensive organisations.

4 Examples

In this section we mention 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 in business and academia. For reasons of space, we give two in detail, and only brief descriptions of the other two. Comments, criticisms and suggestions from readers of this paper are welcomed, as part of the validation process.

4.1 Modularity in phased processing systems

This pattern is 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, rather than writing a new system from scratch.

Name: Externalising an internal representation

Status: Draft: seen several times, but not often enough to be fully confident that this is the right abstraction.

Example: A common example comes from the world of compilers. Most programming language compilers are able to perform optimisations on the output of syntax and semantic analysis, transforming parts of the syntax tree. These techniques evolved some years after most commercially produced Fortran compilers were in widespread use. Such optimisations are computationally expensive and may cause problems in locating runtime errors, so they must remain optional. On the other hand they are incremental, so that new forms of optimisation may be added later. Equally, when compiling for vectorising or parallel computers, suitable restructuring may be done at the same stage, to take advantage of the features of these machines. Most compilers evolved from monolithic programs, where the syntax tree was held in memory, and now use intermediate formats such as triples or quads, written onto temporary files [21].

Context: Technical:

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. This is shown in Figure 2a.

Business:

There is a requirement – typically coming from the need to match competing products – to add new (optional) phases between phases which previously were always called consecutively. It is expected that other new phases may be required in future, for example because of a fast changing market.

Problem: Phases are not currently well encapsulated: wherever two phases are currently consecutive, they always 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. 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. In addition, the current system's functionality must not be compromised, otherwise existing customers may move to competing products.

Solution: 1. Incrementally replace the internal format with a newly defined and fully documented interface format, open to use by new phases. Modify the original first to output the new format as an optional alternative to the current means of sharing the information.

2. Develop the new optional phase using the new interface format as first input and then output. The working of the new phase can be checked by feeding its own input back into itself. At this point, shown in Figure 2b, the original first phase outputs two formats depending on what its successor will be.

3. Modify the old subsequent phase to input the new format. The old format can now be abandoned, and the ability of the old first phase to output the old internal format can be removed, as shown in Figure 2c. The structure of the new system is now modular, with the driver program as a Mediator, as in the design pattern[18].

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.

The original means of communication between the first and subsequent phase can be preserved as an option until the new external representation is fully tested. This avoids compromising existing uses during reengineering.

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 use of an external file, rather than memory, is an option with the new structure. This may be beneficial where memory is at a premium.

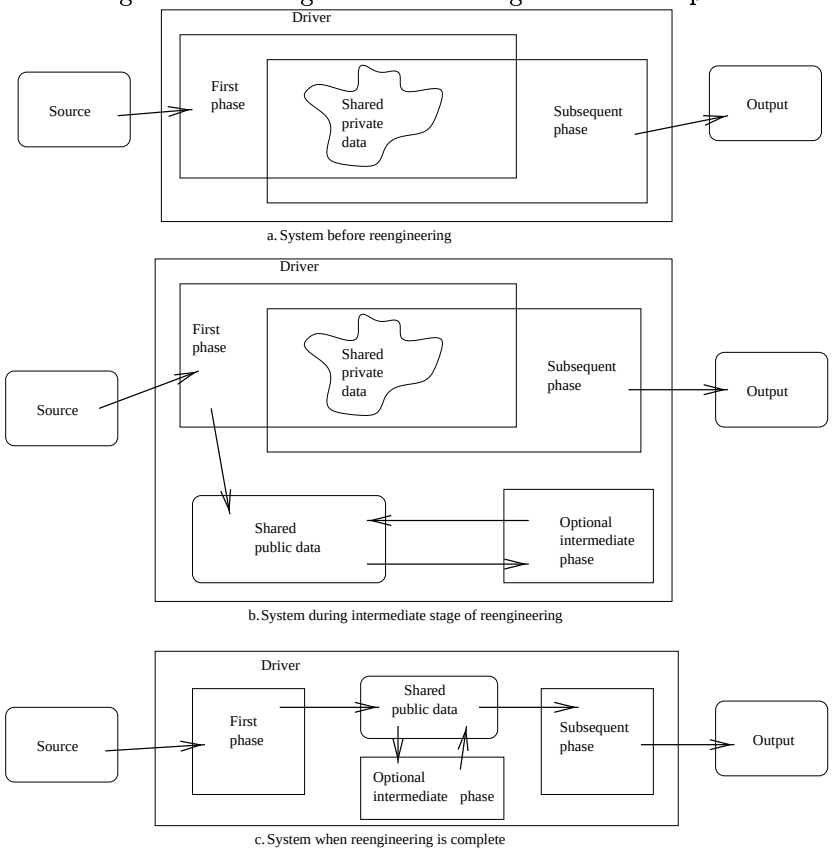
The new architecture may also allow more portability to new back-ends.

Known uses: Two uses were observed personally by one of the authors (Pooley).

1. A commercially marketed Fortran compiler, written at the Edinburgh Regional Computer Centre, was adapted to include optional optimisation phases. Maintaining existing correct behaviour was a major constraint, while fierce competition was a driver for change.

2. During the development of the Integrated Modelling Support Environment (IMSE) as part of an ESPRIT collaboration [22], the graphical interface was supported by a generic platform, known as the

Figure 2: The stages in externalising an internal representation



Graphical Interface System (GIS). This had been developed in-house by ICL before the IMSE collaboration. In opening this front end up to the other eleven partners, ICL developed a shareable text based representation, known as the Graphical Description Language (GDL), to replace a private, internal representation in use originally. This led to successful integration of several new facilities.

4.2 Changing interfaces in a client-friendly way

This example 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 also draws on Stevens' experience reengineering the Edinburgh Concurrency Workbench. This highly complex system had evolved a structure which was clearly far from ideal, but resources limitations made it impractical to impose a new structure and newly designed interfaces in one go.

Name: Deprecation

Status: Folklore: "all experts knows this"

Context: Technical:

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.

Business:

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.

Known uses: Emacs, Java JDK, internal Edinburgh Concurrency Workbench development.

4.3 Other examples

We briefly mention the examples which we have not described in detail here for reasons of space.

- **Portability through back-end abstraction** describes the use of an abstract intermediate language in the process of extending a family of compilers from a single platform to multiple platforms, whilst continuing to maintain the existing products.

- **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. It describes an approach to the problem of reducing dependence on obsolete technology by replacing part of a legacy system with a new system, whilst managing expectations so as to mitigate the risk of dependency on obsolete technology before addressing the long term requirements of an enhanced system.

5 Conclusions and future work

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 must make decisions about reengineering in a reasoned way, taking advantage of the experience of others. Understanding and evaluating the approach requires an appreciation of how such professionals learn to solve problems; a referee has rightly pointed out that there is a wider debate to be entered here. Schön [23, 24] has studied the professional as "reflective practitioner" and there are several interesting points of contact. A patterns approach fits into his observed viewpoint: he writes "When a practitioner makes sense of a situation he perceives to be unique, he *sees* it as something already present in his repertoire". Patterns may help to extend a practitioner's repertoire beyond what has been acquired by direct experience; but of course something is lost too. By discussing (in the Consequences section, in our presentation) the pros and cons of a pattern, the pattern writer attempts to encourage the pattern user to reflect on the problem and its solution.

We can at least begin to compare patterns as an approach to learning reengineering to the only common alternative to learning by one's own experience, namely following a written reengineering methodology. 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 would like in future work to facilitate the writing of patterns by practitioners, perhaps as part of the project review process. This can be seen as an element of a strategy for knowledge management within an organisation; it would also be interesting to explore the effectiveness (or otherwise) of *writing* patterns as a means of learning.

We have proposed some candidate patterns to illustrate the technique and its scope. To take the idea further requires continuing input from many sections of the reengineering community, and this paper solicits more such input. You are invited to consult our systems reengineering patterns Web page:
`http://www.reengineering.ed.ac.uk/`
and to send comments or suggestions to the authors.

6 Acknowledgements

We gratefully acknowledge help from: the participants in the ECOOP Workshop in Object Oriented Reengineering, especially the members of the working group on Reengineering Patterns, whose discussion is reported in [25]; members of the systems-reengineering-patterns mailing list, especially Stéphane Ducasse and Sander Tichelaar; the EPSRC (GR/M02491); the anonymous referees.

References

[1] Rugaber, S. and Wills, L. M. 'Creating a research infrastructure for reengineering.' *3rd Working Conference on Reverse Engineering* (IEEE Computer Society Press, 1996) pp. 120–130

[2] Jacobson, I. and Lindström, F. 'Re-engineering of old systems to an object-oriented architecture.' *Proceedings OOPSLA '91, ACM SIGPLAN Notices*. November 1991 pp. 340–350. Published as Proceedings OOPSLA '91, ACM SIGPLAN Notices, volume 26, number 11

[3] 'Software reengineering assessment handbook v3.0.' available from `http://stsc.hill.af.mil/RENG/`

[4] Alan W. Brown, E. J. M. and Tilley, S. R. 'Assessing the evolvability of a legacy system.' CMU SEI draft white paper, 1996

[5] Jane Ransom, I. S. and Warren, I. 'A method for assessing legacy systems for evolution.' *Proceedings of Reengineering Forum*. 1998

[6] F., H. D. Development of Information Systems (Ronald Press, New York, 1968)

[7] Kempis, R.-D. and Ringbeck, J. 'Manufacturing's use and abuse of it.' *The McKinsey Quarterly*, 1998. **1**, pp. 138

[8] Brodie, M. and Stonebraker, M. Migrating Legacy Systems — Gateways, Interfaces & The Incremental Approach (Morgan Kaufmann Publishers, Inc., 1995)

[9] Stevens, P. and Pooley, R. 'Systems reengineering patterns.' *Proceedings of ACM-SIGSOFT* Foundations of Software Engineering 6. November 1998 pp. 17–23. ISBN 1-58113-108-9

[10] S. Ducasse, R. Nebbe, T. R. 'Type-check elimination: Two reengineering patterns.' presented at EuroPLOP'98

[11] Cunningham, W. 'Episodes: A pattern language of competitive development.' available from `http://www.c2.com/ppr/titles.html`

[12] Coplien, J. O. 'A development process generative pattern language.' *Proceedings of PLoP*. 1995

[13] (administrator), J. C. 'Organizationalpatterns web page.' `http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns`

[14] Appleton, B. 'Patterns for conducting process improvement.' *Proceedings of PLoP*. 1997

[15] Beedle, M. 'Pattern based reengineering.' Object Magazine, 1997

[16] Cinnéide, M. O. and Nixon, P. 'Program restructuring to introduce design patterns.' Technical report, Dept Computer Science, University College Dublin, 1998

[17] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stad, M. Pattern-Oriented Software Architecture – A System of Patterns (John Wiley, 1996)

[18] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design Patterns (Addison Wesley, Reading, MA, 1995)

[19] Meszaros, G. and Doble, J. 'A pattern language for pattern writing.' available from http://hillside.net/patterns/Writing/pattern_index.html

[20] Cunningham, W. 'Tips for writing pattern languages.' available from http://c2.com/cgi/wiki?TipsForWritingPatternLanguages

[21] Alfred V. Aho, R. S. and Ullman, J. D. Compilers, Principles, techniques and Tools (Addison-Wesley, 1986)

[22] Pooley, R. J. 'The integrated modelling support environment.' *Computer Performance Evaluation - Proc. 5th Int. Conf. on Techniques and Tools for Computer Performance Evaluation* (North Holland, 1991) pp. 1–16

[23] Schön, D. A. The Reflective Practitioner: How Professionals Think in Action (Basic Books, 1983)

[24] Schön, D. A. Educating the Reflective Practitioner (Jossey-Bass, 1990)

[25] Stevens, P. 'Report of working group on reengineering patterns.' ECOOP Workshop Reader (Springer-Verlag, LNCS)

Captions:
Figure 1: Decision matrix
Figure 2: The stages in externalising an internal representation