Three Patterns from the ADAPTOR Pattern Language

Alan O'Callaghan
Software Technologies Research Laboratory
SERCentre
Faculty of Computing Sciences and Engineering
The Gateway
LEICESTER LE1 9BH
United Kingdom

+44 116 2551551 x6718
aoc@dmu.ac.uk

Introduction
The patterns below are part of an emerging pattern language called ADAPTOR. The acronym stands for 'Architecture Driven And Patterns-based Techniques for Object Re-engineering'. As the name implies, ADAPTOR is used in projects where the aim is to migrate an existing legacy system to an object-based and/or component-based architecture.

History
The ADAPTOR patterns were originally mined from a series of five different migration projects in different business domains within the telecommunications sector beginning as far back as 1993. Originally, the lessons of these projects were not captured as patterns, but this became a conscious aim from 1995 when, at first, individual *design* patterns were collected in a catalogue. Very quickly it became apparent that other kinds of patterns, notably organisational patterns, were needed to deal with some of the challenges that routinely emerge when an attempt is being made to change an existing, brittle structure into one which is more flexible to change. At the same time interconnections between the individual patterns suggested that there was a possibility of developing less a catalogue of *individual*, "standalone" patterns and more of a pattern *language* whose patterns, when applied in context in an appropriate sequence, could "generate" solutions. The ADAPTOR project became a conscious one from 1996 onwards, though as mentioned above, the patterns were mined from projects from 1993. Incidentally, in its current version, patterns have been mined from further projects in rather more varied sectors such as the defence industry, the oil exploration industry and the retail buying industry.

Refactoring
The realisation that there existed a possibility to explore a pattern language for migration required more than just adding new patterns to the catalogue and rearranging existing links. The original patterns themselves had to be recrafted. A general effect was that existing patterns became much smaller, and apparently simpler (to the extent that some, when viewed in isolation, almost seemed *simplistic*) because some of the ground they had covered was now addressed by newer patterns that were linked to them. The language, even in its current, immature form is sophisticated, even complex, but that complexity is distributed through the language as a whole and is no longer apparent by viewing patterns in isolation. The patterns below are part of that recrafting process which has already seen seven other patterns

workshopped at EuroPlop '99 and EuroPlop 2000. A map of the relationship between those patterns and the three presented below is included at the end of this paper

Pattern template

The Coplien pattern form (Coplien 1995) was adopted to present the patterns in the public domain (some of the patterns are held in different template form in-house by the companies that hosted the original projects). The relevant sections of this template are: *Name*, *Problem*, *Context*, *Forces*, *Solution*, *Resulting Context*, *Rationale*. The attractiveness of this form is that the *Context* and *Resulting Context* sections provide the links in the language, i.e., the *Resulting Context* of a pattern applied in a system should be the *Context* of the next one to be applied, and so on. Coplien's form was developed to present his organisational patterns, of course, and ADAPTOR contains at least design patterns and organisational patterns and, possibly, other kinds of patterns too. Recognising this an additional section, *Classification*, has been added to the template. The emboldening of the *Problem* statement, which is always posed in the form of a question, and the first sentence of the *Solution* section provides thumbnails for shorthand descriptions of the patterns for searchability.

The ADAPTOR approach to legacy system migration

As can be readily seen from examining the patterns below, ADAPTOR's approach to legacy system migration is radically different from 'traditional' approaches which are based on the application of formal methods to existing source code. Legacy systems are, by definition, living systems and we find it no more appropriate to use methods akin to archaeology than we would if they were suggested for the diagnosis of symptoms of ill-health to living organisms such as people. The traditional approaches, in common with all masterplan or blueprint approaches (see Coplien 1999 and Gabriel 2000), seek to systematically exclude the human dimension in favour of automation. In the context of legacy business systems this means two crucial factors are ignored: first, both the explicit and tacit knowledge of the system under redevelopment which is held exclusively in the heads of those who have developed and maintained it are ignored; second, the actual needs of the various classes of user of the system-to-be built (i.e., the 'migrated' system) play no part.

Future work

In placing the human dimension at the centre of its concerns, the ADAPTOR approach to migration is brought closer to piecemeal growth approaches to the construction of large, business systems. In fact many of its patterns, for example ***Modello*** below, are applicable in greenfield development. This realisation has resulted in a new project called Janus in which ADAPTOR is regarded as a subset of an pattern language for the praxis of software architecture. We do not yet know whether the Janus/ADAPTOR patterns that currently exist indicate the feasibility of such a language though we are increasingly confident that that is the case. It is certain that such a language will include many more patterns than are currently included in the language. It is equally certain that the construction of this language will involve the efforts of far many more developers and pattern authors than has been the case to date. The next phase of this work will begin in the New Year with the publication, in book form, of the existing set of ADAPTOR/Janus patterns.

Mercenary Archaeologist

Classification
Role

Problem
How do you deal with information in the existing documentation of a legacy system?

Context
A legacy system is to be migrated to an object or component-based architecture. ***Get the Model from the People*** and related problem-setting patterns have been applied. At best, fragmentary and unreliable documentation of the legacy system is available.

Forces

- Legacy systems worthy of the name add significant value in their current usage, and are often indispensable BUT they are inflexible to the needs of anticipated future requirements.
- 'Good' architecture requires that the nature of the software solution be shaped as far as possible by the nature of the problem BUT existing software assets should not be thrown away needlessly.
- Traditional reverse engineering techniques rest on the notion that source code is the only reliable documentation of a system BUT the function of source code is to instruct the virtual machine, not to represent the problem space.
- Detailed information pertaining to the current state of the legacy system can provide valuable insight into the problem BUT there is a danger of the existing system design overconstraining the new one.
- Documentation can be useful BUT, under deadline pressure, very often it is only source code that is updated and cannot therefore be relied upon to describe the existing state of the system.
- The people involved in using the system, those who have developed it and those who maintain it are the wellspring of the most valuable information about the system BUT detailed knowledge of the intricacies of the current implementation may include redundancies, "dead" data etc., which is outside of their knowledge.

Solution
Hire specialist resources (people and or tools) to recover the information from the code itself, but keep this activity off the critical path of the project.

Resulting Context
Information retrieved from system documentation and/or source code is accepted as input into the requirements engineering and/or specification activities but does not drive the development itself. Scoped by careful problem-setting using means familiar to the development team when it does "greenfield" development, this information can be audited, given its appropriate weight and designated a level of usefulness. The project as a whole

proceeds on lines similar to a greenfield development, and scarce and expensive reverse engineering resources are applied selectively.

Rationale

A legacy system is not a problem, it is a possibly inadequate or misconceived *solution* to a problem. The problem itself cannot be reconstructed from code, only the details of the existing solution can be. Since, by definition, a legacy system migration involves finding a new and, hopefully, better solution to the problem a migration project should begin by using the same kinds of problem-centred techniques as would a Greenfield system. If object modelling is being used this might involve an object model of the problem space, built by using patterns such as ***Get the Model from the People***.In this initial, problem-centred work, the legacy system need only be modelled as a black box providing services.

However it is unlikely that the legacy system can be ignored completely. Firstly, if there is maintenance documentation it may reveal information about domain descriptions, requirements and even software specifications that might not otherwise be readily available. Secondly, a legacy migration always involves keeping some parts of the legacy in the new configuration ('harvesting') and that will involve some form of dependency and constraint analysis in order to identify where surgery can take place to separate it from unwanted and replaceable code. In either case traditional reverse engineering techniques, using formal methods to recover specifications from code, may be appropriate – but only as an adjunct to the main modelling tasks.

This pattern is inspired by, and in some cases can be regarded as a specialisation of, Coplien's ***Mercenary Analyst*** (Coplien 1995) applied to the specific context of legacy system migration.

Reception Committee

Classification
Role

Problem
How do you downstream lessons learned in a pilot project involving ‘new’ technology such as objects or components?

Context
An enterprise had determined upon migrating its software technology base to object or component-based technology. Its current personnel are trained and have high competencies in the ‘old’ way of doing things. A pilot project has been announced in order to begin the process of familiarisation with the new technology. The aim is to migrate the whole enterprise to this new way of doing things.

Forces
- New skills are at a premium BUT so is working knowledge of the existing system
- Where a technology shift means acquiring a new ‘mindset’, it is always useful to hire some people who already have those skills, BUT there is also the tacit business knowledge held by those who develop and maintain the current systems to be remembered.
- It takes longer to train ‘good’ and experienced structured developers in OO and component-based skills than it does novices, BUT in the long term the best technologists will be more productive irrespective of the paradigm being used.
- Mentoring of projects is the best way to upgrade skillsets BUT experienced consultants are expensive to hire if, indeed, genuine experts are available at all.
- Risk management strategies, as well as general cost implications, strongly imply an incremental approach to training/familiarisation in the new technology BUT this can be divisive as those not initially involved fear for their futures.
- Upskilling is a minimum condition of success for a technology adoption process BUT in a buyers’ market, experienced developers with newly acquired skills become targets for headhunters.

Solution
Select a team for qualities that include the ability of the individuals concerned to act as mentors in future projects. Make it clear to the entire enterprise that those involved in the first projects will be expected to mentor and lead future project teams. In short they will become the key personnel in the downstreaming mechanism. This role should be reflected in reward schemes with appropriate status and/or financial incentives.

Resulting Context
The major consequence of the use of this pattern is twofold: first the potential for division between those applying the new technology and the (initial) majority still working with the old stuff is minimised. The enterprise as a whole has a vested interest in both the success of the pilot, and of the pilot project team members in gaining confidence to help disseminate the

new ways of working. The pilot project members themselves are given an incentive to remain with the enterprise after their training/familiarisation period, and overall training costs are lowered by minimising the reliance on hired consultants for training/mentoring. An added, beneficial side effect is that, as opposed to hired experts, internally grown mentors are more likely to interpret the use of the new technology in the context of the enterprise's development culture as they downstream the new stuff. The downside is that this is a slower, if possibly surer, way of a critical mass of developers acquiring the necessary skills than by training them *en masse*. In the meantime (i.e., before the critical mass is reached) all of the forces described above have impact, with possible negative consequences.

Rationale

There is a rich experience of the use of this pattern in industry, particularly as companies that can be considered as early adopters of object technology took up the paradigm. 'Object Centres' often played this role. The pattern can be regarded as a specialized application of two of Coplien's patterns in combination: ***Gatekeeper*** and ***Firewall*** (Coplien 1995).

Modello

Classification
Problem-setting

Problem
How do you visualise an architecture at the beginning of a project, or a fragment of a solution, without overspecifying it?

Context
A new system is being considered, either a greenfield development or the migration of a legacy system to a radically new structure. ***Mile-Wide, Inch-Deep*** is being applied OR a 'tricky' design problem needs visualisation in order to solve it.

Forces
- Maintaining the conceptual integrity of a system requires architectural vision BUT software is inherently not visualizable (Brooks 1986)
- Standard model-driven development often demands 'seamless' iterative changes to a single model through the various phases of development BUT this implies the gradual erosion of the original idea
- Using drawings are a general way of abstracting aspects of difficult problems so that potential solutions can be envisioned BUT overuse of drawings can lead to abstract, unfeasible solutions being produced
- Symbolic modelling (e.g., 'analysis and design' with UML) standards exist for visual modelling BUT these are often tied to heavyweight processes (such as the Unified Process) which mandate drawings for reasons other than visualisation, e.g., specification, documentation, code generation and so on

Solution
Develop a model that projects the architectural vision of the system (or subsystem) to be built, but only to the level of detail that allows the next step to be taken. Where used to envision the architecture, keep the *modello* and maintain it separately from the formal documentation of any models used in the development phases, but require correspondence constraints to be put in place between these models and the *modello*. Where used for visualisation of particular problems, there is no requirement for maintaining the *modello* once it has achieved its purpose. Although formal notations such as UML can be used to depict a *modello* there is no requirement for completeness in any diagrams that describe it because it acts only as a visualisation. The test for its usefulness is the ability of developers to take the next and subsequent design choices based upon it, not its executability, its formal correctness or any other such test as might otherwise be required by the following:
- Specifications of software (either for manual coding or code generation)
- Descriptions of the (built) systems or sub-systems

Resulting Context
As an envisioned architecture, a point of reference is created for all future design decisions in the development process. The model acts as a visualisation of the architecture, nothing more and nothing less. Particularly in the context of ***Mile-Wide, Inch Deep*** (which provides a

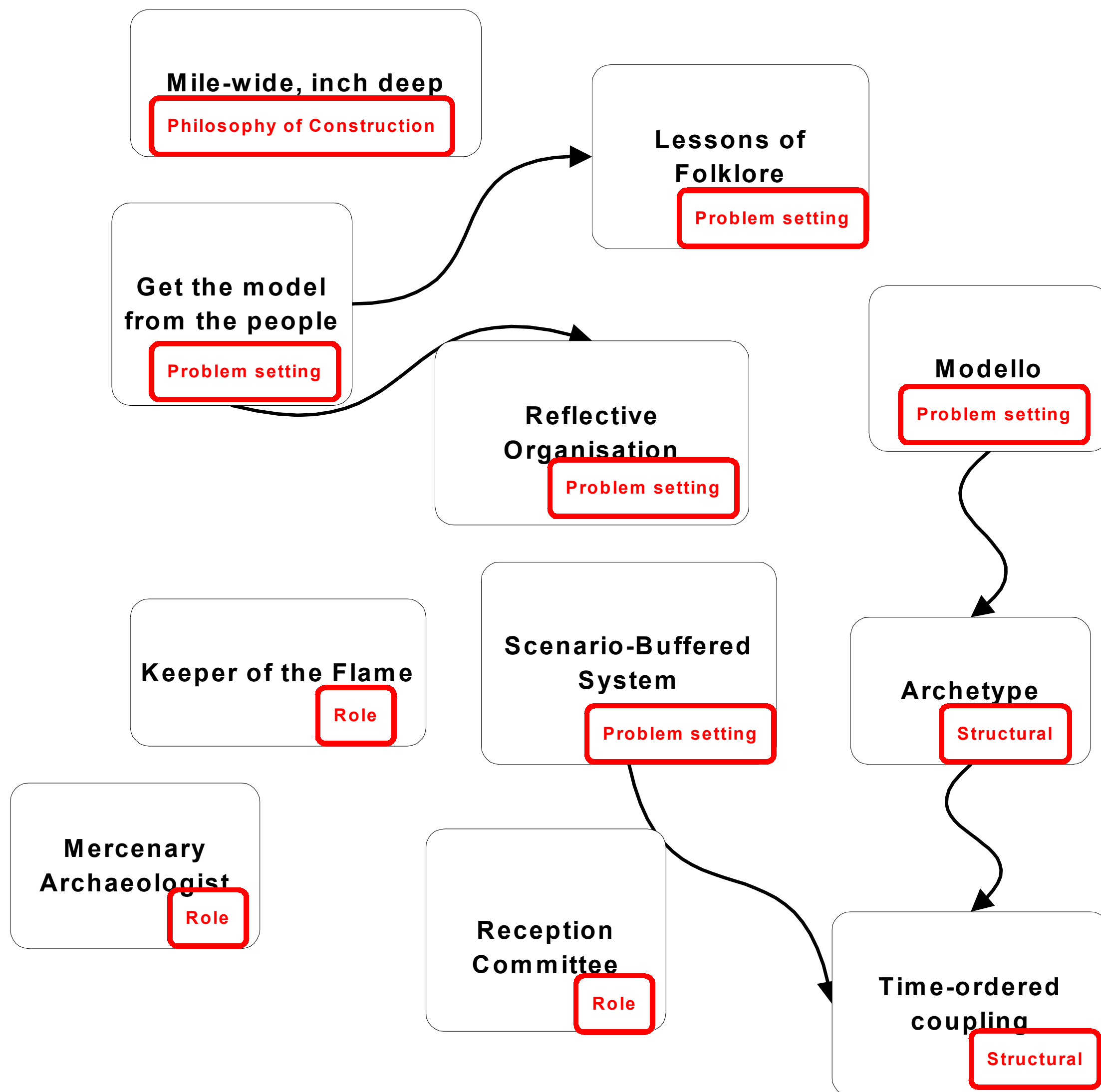
global structure to the evolving system) and of the ***Keeper of the Flame*** role pattern (the Keeper is the role responsible for facilitating interpretation of the architecture), changes to the architecture necessitated by detailed design decisions or by external constraints can be negotiated as the system's structure is specified.

In more general use ***Modello*** frees programmers and other developers from the constraints of notation semantics, the need for formal correctness etc., in producing drawings primarily for their own use and designed only to see them through to the next step in their work.

In general the pattern separates out drawings that are needed, or may simply be useful, to visualize solutions from those needed to provide detailed specifications, descriptions etc., in line with formal Quality Assurance or other standards. It complements ***Mercenary Analyst*** (and ***Mercenary Archaeologist***) and lays the basis for their use.

Rationale

Modello was the term for the scale model often presented by, for example, renaissance architects to the panel of judges in open competitions for the contracts to build churches, palaces, cathedrals etc. The adopted scale model would often be placed in a knave or chapel as a cathedral was built around it both as reference for the various kinds of builders working on it, and as an advertisement to the general public as to what it would eventually look like. It was a visualisation of the building that represented a *requirement* rather than a detailed specification of the construction. Indeed, in the famous case of the dome of Florence's cathedral the engineering problem of how to construct it went unsolved for many years before Filipo Brunelleschi presented his famous, innovative solution despite being reflected in the original *modello* of the cathedral for decades beforehand.



ADAPTOR patterns

The above patterns have been workshopped at EuroPlop conferences between 1999 and 2001. The wavy arrows indicate links that are be populated by other patterns (as yet unworkshopped, or in other public domain catalogues and languages). Each has been 'rubber stamped' with its classification. Note that *Lessons of Folklore* was originally entitled *Pay Attention to the Folklore* and *Scenario-Buffered System* was called *Buffer the System with Scenarios* when first workshopped.

References

(Brookes 1986) Brooks Jr., F.J. 1986. “No Silver Bullet” in F.J. Brooks Jr. *Mythical Man-Month*. Reading, Mass: Addison Wesley
(Coplien 1995) Coplien J. O. 1995. A Generative Development-Process Pattern Language in J.O. Coplien and D.C. Schmidt (eds.)1995. *Pattern Languages of Program Design*. Reading, Mass: Addison Wesley
(Coplien 1999) “Re-evaluating the architectural metaphor – Towards piecemeal growth” in IEEE Software. Special issue on Software Architecture. October 1999. pp.40-44
(Gabriel 2000) Gabriel R. “Mob Software – the erotic life of code”. 2000. Essay presented at OOPSLA 2000. Minneapolis, Minn.