An STL Primer and a more detailed consideration of idiomatic use of the STL, together with papers on Smart Pointers in C++ and Evolutionary Delivery are also available at this site. For more information on the author, please take a look at my software and home pages.

The Role of Patterns in Enterprise Architecture

David Harvey
April 1998
Contact
http://www.davethehat.com/

This paper formed part of a presentation at a Unicom seminar on Patterns for Distributed Computing held in London on 18 May 1998

Why Architecture?

Systems designers and implementers have a hard time. We're expected to build more and more complex systems, addressing ever-changing business requirements (so the systems must be flexible), capable of delivering timely information (we want to know the state of our business as of *now*, not yesterday), to increasingly large numbers of users (let's put the system on the intranet/Internet). We're being asked to deliver these systems quicker than ever (in spite of the software industry's much publicised poor record in this area), into an environment in which software and hardware are changing at an increasing rate, and while we're coping with this, let's deal with the millennium and EMU.

My goal here is to encourage you to share my belief that doing architecture right is key to solving these problems, and to building the systems we need for the next century. Patterns are a powerful means of promoting architectural vision, in software as well as in building, so this paper will point out ways in which patterns contribute to the architectural initiative.

I'm not going to argue over terms: although an overused word, "architecture" will have to do to represent both the essential large-scale structural and dynamic characteristics of systems, and the distinctive set of conditions and processes by which the commission, design, build and lifetime of such systems is achieved. Fred Brooks stresses the importance of *conceptual integrity* in determining architectural quality, and these two points encompass both the description of that integrity as a property of the artefact, and the organisation, rules and rationales that help us realise it.

Architectural complexity is closely dependent on scale. Clearly, even the smallest system can be said to have an architecture (just as a shed or dovecote can be talked about in these terms). But the systems we're concerned with are more complex than this. While there are valid points to be made about the organisation of single-user, single-process stand-alone programs, these do not come within my scope. It's important to distinguish between *enterprise architecture* (which is concerned with the interaction and evolution of multiple software systems supporting an enterprise's changing business needs over time), *system architecture* (specific to large systems composed of multiple components running on different processors) and *component architecture* (proper to the architecture of an individual component).

Enterprise architecture offers a lifeline. The approach allows us to think at a high level and for an extended timescale about IT support of business, from a logical level (services to support information and processing requirements of business processes) down to a physical implementation (the choice of *this* network, *these* servers, *this* strategic middleware).

Why Patterns?

In the development community patterns have become a key component in the vocabulary of design and implementation. By now, the history is well-known - first devised by the building architect Christopher Alexander as a means of capturing the qualities of successful communities and buildings, their adoption by the software community through the support of Kent Beck, Ralph Johnson, Erich Gamma, John Vlissides, Bruce Anderson and many others has been a defining characteristic of the field in the 1990s.

With the publication of the catalogue of Design Patterns [GAM95] by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, the ideas reached a wider audience of developers, but the success of this book has had the consequence of focusing attention on patterns as solutions to *implementation* design problems – the level of component architecture described above.

Patterns, Pattern Catalogues, Pattern Languages

Patterns come in collections of different sizes. Much of the literature exists in the form of stand-alone single patterns. These constitute useful descriptions of point solutions to problems. A pattern catalogue is essentially a collection of such patterns, usually united by an external motivation such as level of applicability [GAM95] or programming language [COP92]. Such catalogues play the role of architects and engineer's pattern-books, promising quick deployment of structures with known properties into the systems we're building.

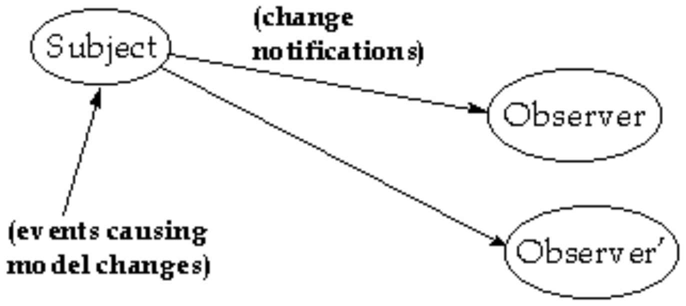
One danger with both single patterns and catalogues is that of over-eager and ill-considered use. Give a child a hammer, it is said, and suddenly everything is a nail. The fundamental problem of the catalogue of patterns is that essential context is missing from the interrelationships of the patterns themselves. For me, the power of patterns lies in their ability to capture a larger body of practice *and the vision which motivates that practice*. The pattern language is the medium best suited to expressing this power [HAR98].

A pattern language is in essence a large cognitive map which expresses the connections between details at all levels of scale. The extent to which you buy in to the language's vision has more to do with the extent to which that map overlaps and completes your own world view than with the technical applicability of individual patterns. This *aha* experience is characteristic of many people's first introduction to patterns in whatever field.

Some recent writers have explicitly tackled software patterns at a higher level of architectural scope, either in passing or as an explicit motivation. Individual patterns in these collections are interesting enough, but there is nothing, for example, in [BUS96] that lives up to the book's title – *Pattern-Oriented Software Architecture* – or subtitle – *A System of Patterns*. The patterns in [MOW97] are too sketchily described, and once again do not cohere as a vision of what it is that pulls an architecture together. At least they do start to address the inherent difficulty of distributed architecture.

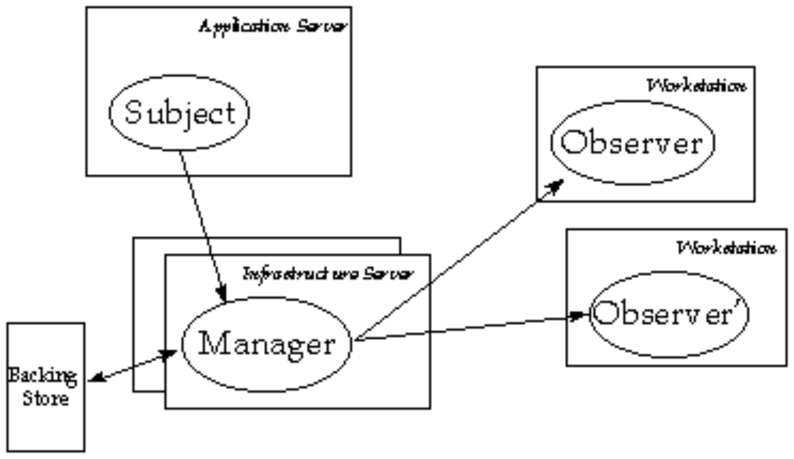
Why is Distribution Hard?

Consider a simple but pervasive pattern such as Observer, a powerful and easily-grasped abstraction of dependency which has been implemented many times over in libraries and languages:



In the context of a single thread executing on a single machine, we can make all sorts of assumptions about lifetimes and identities of objects participating in this pattern. How many of these assumptions hold if we want to place these objects in different processes, and on different machines? What does it take to distribute this pattern?

In a distributed environment, having each subject maintain its own relationships with observers introduces large numbers of fragile interdependencies, so typically this pattern is enhanced by introducing a manager object, and to maintain state and event information this will usually be associated with backing store of some description. This manager is clearly a potential bottleneck, not to mention a single point of failure, so it is often replicated, using the same or a replicated backing store.



If subjects or observers are transient, then there may be no particular issues with starting or restarting the processes hosting them. If, however, a subject is logically persistent, then restarting its process will necessitate some way of rebuilding the last stored state of the subject. Similarly, if observers need to maintain the illusion of persistent connection to their subjects, on restarting there had better be some mechanism for synchronising the observer's view of the subject with its current state.

We need to consider the issue of rendezvous. A strict implementation of observer based on object identity will need to use distributed object identities. These will typically be under the management of an ORB or similar technology, but the instantiation of these objects on different machines makes us ask several questions. Can a logically persistent subject be observed if it does not currently exist as an object in a process? What happens when a subject fails or terminates? When an observer fails or terminates? Or when a subject or observer starts restarts? Can we even *tell* if a subject, observer or manager disappears?

Ordering becomes a significant issue. If a subject generates events in a given order, it is usually important that order, but this is by no means guaranteed in an environment in which the physical routing of network messages may be different for each event.

In this pattern as it stands, and assuming updates to the subject occur sequentially, consistency may not be a significant problem. The focus of the pattern is the effect and propagation of change rather than its co-ordination. However, if the subject is referenced by many updating objects some kind of transactional policy is required to prevent inconsistent views in the observers arising from jumbled event sequences.

The role of patterns

Patterns in coding

Also known as idioms, these low-level patterns grow from the lore of programming in a particular language or environment. They tend to be highly specific to those environments, although patterns can spread. Model-View-Controller, perhaps the first software artefact to be recognised and generalised as a pattern, started life as a mechanism for decoupling model and graphical presentation in the Smalltalk environment.

Patterns in component design

This is the level at which most developers and designers know about and apply patterns. While undoubtedly true that design patterns are a powerful medium for capturing and conveying design expertise, the packaging of design patterns in the literature has fostered an off-the-shelf approach, which sometimes leads to over-zealous application of inappropriate patterns.

Patterns in architectural design

Literature and usage here is less well established. This is largely because implementation at the architectural level is hard, with the number of people working specifically at this level being correspondingly smaller. In spite of the fact that a few books purport to deliver architectural patterns, I don't think yet we've had time to consolidate the experience of architecture into a set of patterns for architectural design. The good news is that usually we can buy the implementations of patterns at this level in the form of middleware products –

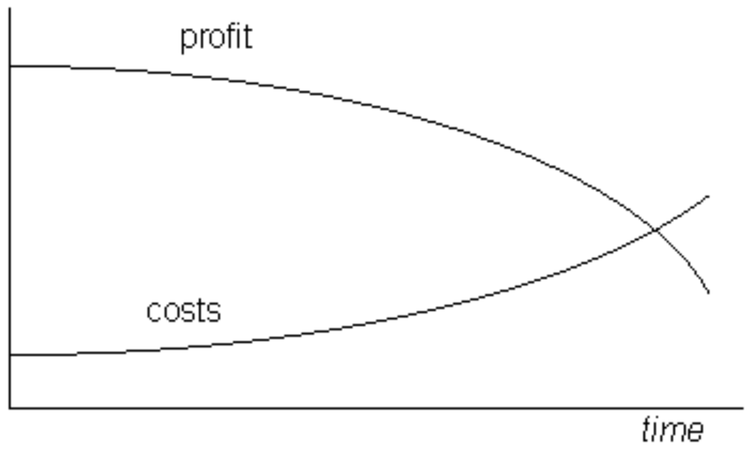
messaging technologies, ORBs, these and others relieve us of the task of writing large amounts of infrastructural code simply to deal with communication and distribution. The best of these products are clearly built around key patterns for distribution. Understanding these and the way they are implemented leads to effective use of these products.

Patterns in architectural process

The only way we're going to ensure conceptual integrity in a process is by having a common understanding of the sorts of decisions and actions appropriate to a particular stage and level of that process. In practice this will only happen if all involved know why this action is appropriate in the context. A pattern language is a powerful means for capturing motivation and rationale in addition to activities.

How to Get Ahead in Architecture

IT's record of delivery, especially for grand plan projects, has not been good. What makes development at an architectural level important is the continuing pressure on businesses to increase flexibility and reduce costs. In many areas of commerce, competition is reducing profit margins while the cost of running and maintaining business processes and their supporting systems rises. If nothing changes, it's easy to see that we reach a point at which we start losing money:



Inefficient business processes often arise directly from the quirky interactions of multiple disparate systems. Introducing new point solutions in such a scenario can offer only localised and temporary respite. In effect, this amounts to replacing one process-constraining system with another, around which new and equally localised processes will soon start to ossify, perhaps even accelerating the convergence.

Selling the architectural initiative to business and user communities is hard. Whether arising within a business from a process engineering exercise or driven by acquisitions and mergers, the software systems being proposed to support new processes are coming under ever more critical scrutiny. Fundamentally, sponsors need convincing that what we're proposing will work, will be delivered, and will support a business through subsequent change.

Patterns can play two fundamental roles in establishing a distributed software architecture:

- ***Energising the process*** - succeeding in introducing distributed architecture is far from easy. It involves difficult issues beyond already-complex technical ones. A pattern language for architectural introduction serves as a manifesto for the way the project is established and run. Work is in progress to define such a language [HAR98].
- ***Aiding the implementation*** - at the technical level, understanding the patterns behind strategic middleware (whether bought-in or developed in-house) makes effective use of that middleware much more likely. Developers work with, rather than against the capabilities of architectural software: those responsible for deployment and support also have a fast-track route to the sort of mental model required for seamless integration and diagnosis.

Neither of these roles arises directly from the notion of patterns as off-the-shelf solutions to design and implementation problems. Without playing down too much this aspect of their usefulness, it seems to me that patterns offer more than short-cuts to code, and at a higher level. By providing a succinct and commonly understood abstraction of the dynamics of interaction they play a key part in enabling effective design and development of components. And by motivating a vision of what an architectural process can be, they provoke us into taking seriously the conditions which must hold if building such systems is to be possible. As many of us have discovered, the architectural initiative is rarely undertaken in an entirely co-operative environment, and is never politically neutral.

References

[ALE77] Alexander C (1977). A Pattern Language, Oxford University Press

[BAS98] Bass L and Clements P and Kazman R (1998). Software Architecture in Practice, Addison Wesley

[BRO95] Brooks F P (1995). The Mythical Man-Month (Anniversary Edition), Addison Wesley

[BUS96] Buschmann F and Meunier R and Rohnert H and Sommerlad P and Stal M (1996). Pattern-Oriented Software Architecture: A System of Patterns, Wiley

[HAR98] Harvey D (1998). How to Get Ahead in Architecture, workshop session at Object Technology 98 conference

[MOW97] Mowbray T J and Malveau R (1997). CORBA Design Patterns, Wiley

[PLOP1] Coplien J and Schmidt D (Ed) (1995) Pattern Languages of Program Design, Addison Wesley

[PLOP2] Vlissides J and Coplien J and Kerth N L (Ed) (1996) Pattern Languages of Program Design 2, Addison Wesley

[PLOP3] Martin R and Riehle D and Buschmann F (Ed) (1998) Pattern Languages of Program Design 3, Addison Wesley

[SHA96] Shaw M and Garlan D (1996). Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall