Legacy System Anti-Patterns and a Pattern-Oriented Migration Response

Anthony Lauder and Stuart Kent

Computing Laboratory
University of Kent at Canterbury
Canterbury, Kent, CT2 7NF, UK
Anthony@Lauder.u-net.com
S.J.H.Kent@ukc.ac.uk

Abstract

Mature information systems grow old *dis*gracefully as successive waves of hacking result in accidental architectures which resist the reflection of on-going business process change. Such petrified systems are termed legacy systems. Legacy systems are simultaneously business assets and business liabilities. Their hard-won dependability and accurate reflection of tacit business knowledge prevents us from undertaking green-field development of replacement systems. Their resistance to the reflection of business process change prevents us from retaining them. Consequently, we are drawn in this paper to a controlled pattern-oriented legacy system migration strategy. Legacy systems exhibit six undesirable anti-patterns. A legacy system migration strategy must focus upon the controlled elimination of these anti-patterns by the step-wise application of six corresponding desirable patterns. Adherence to this migration strategy results in adaptive systems reflecting purposeful architectures open to the on-going reflection of business process change. Without such a strategy there is a very real danger that legacy system migration will occur all too literally. That is, the old legacy system will be migrated to a new legacy system albeit it one using the latest buzzword-compliant technology.

1. Legacy Systems: Assets and Liabilities

Organizations deploy information systems to support their business processes. As circumstances change, organizations adapt their business processes in response. Consequently, the supporting information systems must be adapted to track this business process change. Unfortunately, mature information systems tend to grow old *dis*gracefully. Although an information system may begin its life with a flexible architecture, repeated waves of hacking tend to *petrify* mature information systems resulting in inflexible *accidental architectures*. Such architectures emerge "… more from the ingenuity of the systems developers' ability to 'warp' the code than from any purposeful design. As successive waves of hacking warp the information system further and further, its accidental architecture becomes more and more convoluted, and the intellectual capability of the developers to identify and apply clever hacks diminishes to a point at which the system is declared un-maintainable." [Lauder and Lind, 1999c]. This process of petrification, then, makes it increasingly difficult to modify a mature information system to reflect on-going business process change. A system which has undergone petrification is termed a *legacy system* [Brodie and Stonebraker, 1995].

Organizations are afraid of keeping their legacy systems, since maintaining them is a significant drain on the organization's resources. They are, however, also afraid of replacing them since those legacy systems have undergone years of debugging effort. As a consequence of this debugging effort they have grown to support not just explicitly understood business processes but also *tacit business knowledge* central to the smooth operation of the organization yet implicitly rather than explicitly acknowledged in the day to day activities of the business. A legacy system, then, is both a business asset and a business liability [Lauder and Lind, 1999c]. They are assets in sense that, through years of debugging effort they have grown to reflect essential tacit knowledge. They are liabilities, however, in that it has become increasingly difficult to adapt them to reflect ongoing business process change. To support a living [Morgan, 1996], learning [Senge, 1990], organization successfully, we must maintain the hard-won reflection of tacit business knowledge which makes legacy systems into assets, whilst simultaneously striving to eliminate the resistance to reflecting on-going business process change which makes those same systems into business liabilities.

1.1 Accidental Architecture Anti-Patterns

We have been commissioned by a mid-sized software development organization to examine their legacy systems and determine what undesirable properties underlie their resistance to reflecting on-going business process change. Having examined these systems, it has emerged that their accidental architectures are characterized by a number of undesirable properties, which can be grouped into six general "anti-patterns":

1. *Ball and Chain:* several of the systems are tied heavily to operating-system-specific facilities, hindering their portability to other platforms.
2. *Tower of Babel:* the systems were typically developed using languages that cannot communicate with one another easily, and thus many of the systems cannot inter-operate.
3. *Monolithicity:* the systems are essentially monolithic, and resist componentization and, thus, reuse.
4. *Gold in them thar Hills:* the code is typically the only repository of domain expertise, and that expertise is scattered throughout the code. This results in both a lack of understanding of the underlying system requirements, and a considerable resistance of the code to safely reflect on-going business process change.
5. *Tight Coupling:* the parts of a legacy system tend to be tightly coupled to one another, thus making them extremely sensitive to changes in each other. This hinders maintainability, adaptability and reuse.
6. *Code Pollution:* the problem-domain-oriented code within a legacy system tends to be wrapped in guard-code that enforces implicit dependencies throughout the system. This guard-code pollutes the code space hinders maintainability, adaptability, understandability, and reusability.

2. Purposeful Architecture Patterns

It is often tempting to throw away a legacy system and write a replacement system from scratch. Unfortunately, the legacy system itself is often the only source of recorded domain expertise. The justified fear of organizations is that a new information system replacing a legacy system may well accurately support the knowledge of which a business is explicitly aware, but will miss – and hence be unable to support - the tacit knowledge implicit in the organization's business processes. The fear, then, is that tacit knowledge will not be reflected in a replacement system until that system has undergone a prolonged and painful debugging process after it has already been deployed and its weaknesses are highlighted via its failure to support the user. The organization has already undergone this pain in the long-term debugging of the legacy system, they do not want to go through it again [Lauder and Lind, 1999c].

Consequently, we propose a more cautious, phased migration strategy from legacy systems to their replacements. When adopting a migration strategy towards new information systems to replace legacy systems we must address the anti-patterns detailed above, otherwise there is a very real danger that legacy system migration will occur all too literally. That is, the old legacy system will be migrated to a new legacy system albeit it one using all the latest buzzword-compliant technology. To work towards the prevention of today's new information systems from becoming tomorrow's legacy systems, we propose six desirable patterns which directly address the six anti-patterns detailed above and thus result in *purposeful, rather than accidental, architecture*. The basic philosophy of our approach is to address each of the six anti-patterns, detailed above, as a separate concern. Thus, our six desirable patterns form a progressive response to the legacy system migration challenge:

1. *Portability Adapter:* construct a portable Fundamental Services Layer (FSL) which provides a platform independent interface to operating system facilities.
2. *Babel Fish:* construct a middleware bridge to each of the legacy systems.
3. *Virtual Componentization:* in the first place, decide what components would ideally exist in each monolithic legacy system, then implement a Facade [Gamma et al. 1995a] on top of each legacy system which provides the illusion of these components already being in place. Over time, real components can be substituted for the virtual components without mandating changes to the interface.
4. *Gold Mining:* This is by far the most difficult of the six patterns to implement. Gold mining is concerned with re-constructing and documenting the requirements captured in legacy system code itself, and taking these mined requirements as the starting point for the negotiation of new requirements. The hope is that the successful completion of phases 1 to 3 will give us enough breathing space to take a more leisurely approach to this challenge.
5. *Implicit Invocation:* to eliminate tight coupling between parts of a legacy system we need to abandon explicit invocation, wherein parts are explicitly aware of one another and invoke known methods or functions on one another, and instead adopt a model of implicit invocation wherein components are decoupled.
6. *Protocol Reflection:* rather than wrap guard code around methods or functions, reflect collaboration protocols explicitly.

3. On the Outside Looking In

We can eliminate the first three legacy system anti-patterns without making major changes to the legacy systems themselves. Addressing the first three anti-patterns increases a legacy system's portability and interoperability and thus gives the impression that migration has occurred already. This is an appropriate interim solution to pressing legacy system concerns, but does not address the fundamental issue of resistance to reflecting business process change. That is a far more demanding issue, and is focus of latter three anti-patterns and their corresponding responsive patterns.

3.1 Portability Adapter: a response to Ball and Chain

When an information system makes direct use of the services provided by an operating system, it risks becoming chained (as in Ball and Chain) to that operating system. This hinders significantly the potential portability of the information system to other operating system platforms. To address this issue, the organization we have been working with has traditionally added conditional code throughout their applications to make different calls based upon whichever operating system they are compiled for. This conditional code, although effective, has two demerits: Firstly, it complicates the readability of the application code and secondly it requires potential modification across the whole of the application whenever a new operating system is ported to. We have proposed an alternative solution, which simplifies the application code and localizes porting changes to a single location. Our proposal, which has been accepted and utilized by our industrial sponsor, is to develop a portability adapter, which we term the Fundamental Services Layer (FSL).

The FSL provides a single consistent interface to common operating system facilities across all operating systems. The application does not make any direct calls to the operating system, rather it makes all calls through the FSL. The FSL then encapsulates appropriate conditional code with translates calls to its standard interface into corresponding calls across the range of supported operating systems. Porting the FSL to new operating systems, then, implies the portability of the legacy systems to those operating systems without requiring any code changes to the legacy systems themselves. All portability issues are now isolated to the FSL, freeing the maintainers of the legacy system and the developers of replacements systems to focus upon application-oriented issues. We are building the FSL on top of Doug Schmidt's ACE toolkit [Schmidt, 1999].

3.2 Babel Fish: a response to the Tower of Babel

Our industrial sponsor has legacy systems developed in COBOL, ESQL, C++, Java, BASIC, and a variety of other languages. Systems developed in the same language tend to be able to inter-operate with relatively little difficulty. Systems developed in different languages tend to resist inter-operation. It is a major challenge, for example, for our sponsoring organization's COBOL and BASIC applications to talk to one another. Traditionally, the sponsor has addressed this Tower of Babel anti-pattern via the development of ad-hoc and frequently rather complicated one-to-one connections between the legacy system applications. Maintenance, development, and use of these bridges have frequently proven frustrating and time consuming. Our alternate strategy has been to construct a middleware bridge to each of the legacy systems. By developing a single bridge to each legacy system, and allowing all languages access to that bridge, we no longer require ad-hoc interconnections across diverse systems. We term such a bridge a Babel Fish. To implement Babel Fish bridges, we are working with the sponsor to develop a CORBA IDL [Henning and Vinoski, 1999] interface for each of the legacy systems, enabling them to inter-operate. For those systems developed in languages for which CORBA offers no mapping, we will construct an Adapter [Gamma et al. 1995] to the legacy system in a CORBA-compliant language (C++) and build an IDL interface for that Adapter.

3.3 Virtual Componentization: a response to Monolithicity

When a system is constructed from large monolithic "chunks", understandability, reliability, and reusability suffer. The solution to this problem is, of course, to utilize manageable focussed software components. However, switching to a completely component-oriented architecture in a single step is inherently risky. A more cautious approach separates two concerns: the interface and the implementation. We propose, then, a two-phase strategy: the first phase focuses upon working out what components would ideally exist, establishing component boundaries, determining their interfaces, and creating a set of corresponding virtual component interfaces on top of the existing legacy system. A virtual component interface, implemented as a Façade [Gamma et al. 1995a]on top a legacy system, provides the illusion of components already being in place. Over time (the second phase), real components can be substituted for the virtual components without mandating changes to the interface. Note that this second phase will usually be undertaken concurrently with the application of the latter three patterns detailed below.

4. On the Inside Looking Out

Once the external view of the legacy system is in place, we can turn our attention to the real challenge, which is the elimination of the resistance to the reflection of on-going business process change. This is a process of replacing the legacy system's accidental architecture with a purposeful architecture whilst maintaining an accurate and complete reflection of the tacit business knowledge embedded within the functionality of the existing legacy system.

4.1 Gold Mining: a response to Gold in them thar Hills

To understand organizations and their supporting information systems, we have adopted the Language Action Perspective (LAP) [Goldkuhl et al. 1998b; Goldkuhl et al. 1998b]. There are many branches of LAP, all bound by the underlying idea that actors communicate with purposeful intent to express needs, align goals, coordinate activities to satisfy those goals, and build enduring personal relationships. In the Business Action Theory (BAT) [Goldkuhl, 1996] [Goldkuhl, 1998a] branch of LAP, this flow from the expression of needs to the enrichment of personal relationships is captured in the *BAT-transaction pattern*. In the BAT-transaction pattern particular emphasis is placed on the process of negotiation that underlies exchange agreements between an organization and its stakeholders. The important point is that business processes fulfill contracts negotiated between and committed to with stakeholders. Since negotiation is perpetual, new contracts emerge continually and, as a consequence, business processes must change to reflect those new contracts. Information systems supporting these business processes must then be adapted to reflect this change.

The BAT transaction pattern helps us to understand the domain in which the information system participates. To manage the legacy system issue, then, we recommend several waves of participatory workshops, wherein the BAT transactions implicit in a legacy system are first reconstructed and refactored, and then taken as the basis for the negotiation of requirements for the development of a replacement information system.

Underpinning our strategy of LAP-oriented information systems development, then, are three sets of participatory workshops (ontology, requirements, and development). Since a workshop is itself a business process, all of the workshops described here follow the BAT transaction pattern. Each workshop begins with a search for appropriate stakeholders (affected actors) to bring to the table. The stakeholders in a workshop then undertake a process of negotiation to reach a shared commitment to some common output. The final part of each workshop is where the participants actually produce that output in fulfillment of the negotiated commitment. The type of output produced is dependent upon the type of workshop.

- **Ontology Workshops**

Ontology workshops focus upon negotiation of and commitment to a mutually acceptable business *ontology* shared by all stakeholders. An ontology [Uschold and Gruninger, 1996] is a shared understanding committed to by a group of actors as a common perception of some domain of interest. In ontology workshops, stakeholders negotiate, commit to, and produce a shared understanding of the business processes upon which information system requirements will be based. There are three types of successive ontology workshops: gold mining, process reconstruction, and ontology refactoring.

When developing information systems to replace legacy systems, the legacy systems themselves are mined for the capture and explicit articulation of the requirements implicit in their current functionality. These (often-tacit) requirements form the hidden gold that makes legacy systems into business assets. Consequently, we term the process that leads to their discovery *gold mining*. Gold mining involves intensive code-inspection workshops alongside the maintainers of the legacy system. The result of gold mining workshops is a deep understanding of the individual collaborations in which the legacy system participates.

Process reconstruction workshops essentially refine the legacy system requirements which emerged from gold mining, reorganizing them to reflect abstractions common to the stakeholders, accommodating current business processes not adequately supported by the legacy system, and renaming areas where more appropriate business terms exist. In effect, this is a refiltering of the requirements and accommodation of current business processes from the stakeholders' perspectives – expressed in stakeholder terms and using their abstractions rather than the terms and abstractions of the legacy systems implementers.

Ontology refactoring workshops, take these reconstructed business processes as a statement of where the business currently is. A prerequisite is that the organization has reflected upon and reached a shared understanding of its goals and the current problems facing the business which prevent those goals from being realized [Goldkuhl and Röstlinger, 1993]. Possible solutions to the problems that the business is currently facing are modeled and discussed. In light of these discussions, a newly negotiated business ontology is constructed which is a transformation (a reshaping, or refactoring) of the current business processes - respecting the transactional contract negotiations, commitments, and fulfillments embedded within them - with an eye on openness to change to accommodate the negotiated problem resolutions.

- **Requirements Workshops**

Requirements workshops take the new business ontology as the starting point for the negotiation of requirements for the supporting replacement information systems. Particular emphasis is place on determining the boundaries of the information system within the business ontology; asking which areas it will support and which roles it will fill in the support of those business areas. It is important to include the developers of the information system in these workshops to get them talking to other stakeholders - to understand the system from the position of the stakeholders and the support roles of the information system to the business. This both teases the developers away from a technology-centered solution - emphasizing requirements-first - and encourages an on-going dialog between the maintainers of the system and those affected by it. Furthermore, since the developers will have a very real understanding of just what is feasible in the realization of an information system, their participation can help to keep any unrealistic information system ambitions of over-enthusiastic stakeholders grounded in what is practical and achievable.

- **Development Workshops**

Developers will then withdraw to their own development workshops - where system designs and prototypes are negotiated, committed to, and produced and ultimately a satisfactory implementation is delivered, using the requirements emerging from requirements workshops as a starting point. Naturally, the developers will probably hold further requirements workshops with the other stakeholders where ambiguities are clarified, prototypes are evaluated, and plans altered as necessary [Ehn, 1988].

For further details on these participatory workshops, the reader is referred to [Lauder and Lind, 1999c]. A major benefit of participatory workshops is that they lead an organization to think about change before it creeps-up on the business and on the maintainers of information systems as late and unexpected requirements changes mandating urgent system hacks. When an organization manages it change purposefully it is able to determine and accommodate the impact of change to the captured business ontology and the information systems that support the business. The key here is an on-going commitment by the business to undertake a perpetual process of reflection and negotiation and the explicit capture of the resultant business ontology. The elimination of legacy system liability, then, results from purposeful reflection and navigation of the business. That is, from the installation of and perpetual adherence to a learning process.

This participatory process of active and joint reflection - armed with a shared business ontology - is of major benefit to an organization even when a resultant information system is intended. Future business process change can then be the result of purposeful reflection through successive waves of participatory workshops. Participatory workshops, then, should be seen as a regular and on-going commitment to a purposeful steering of the business.

4.2 Implicit Invocation: a response to Tight Coupling

Different parts of an information system tend to invoke one another's methods or functions directly. In the legacy systems we have examined, the caller usually explicitly identifies both the part being called and the method of function being invoked. This type of invocation is termed *explicit invocation*, and it results in a tight coupling between the pieces of the information system. This tight coupling makes tightly coupled pieces sensitive to changes to one another and thus increases the likelihood of propagated side effects resulting from what should be localized code changes. This, in turn, increases the potential for the introduction of unanticipated and subtle bugs, and reduces the level of confidence of developers that their changes are free from harm.

To eliminate tight coupling we need to abandon explicit invocation and instead adopt a model of *implicit invocation*. In implicit invocation components register their interest in certain events with a broker. Other components broadcast events through that broker. The broker then propagates those events onto components that have registered interest in them. The broker forms an intermediary between collaborating components, hence resulting in component de-coupling. This reduction in inter-component dependencies enhances component adaptability, maintainability, and reusability. Implicit invocation underlies a number of recent design patterns (Observer [Gamma et al. 1995b], Multicast [Vlissides, 1997], Event Notification [Riehle, 1996], Reactor [Schmidt, 1995], etc.) and a number of commercially available products (Rendezvous [Tibco, 1999], SmartSockets [Talarian, 1999], iBus [SoftWired AG, 1999], etc.).

The authors have implemented an implicit-invocation-oriented technology, named *EventPorts* [Lauder, 1999a; Lauder and Kent, 1999b]. A component developed using EventPorts exports an *Interface* and an *Outerface*. A component receives events from other components through its interface and sends events to other components through its outerface. Interfaces and outerface consist of dynamically configurable EventPorts. There are two types of EventPort: interfaces consist of *InPorts,* via which a component receives events from other components, and outerfaces consist of *OutPorts*, via which a component sends events to other components.

Each of a component's InPorts can be dynamically attached to and detached from any number of OutPorts on any number of other components at a given time. An OutPort acts, in effect, as a broker that registers event interests and receives and forwards messages according to those registered interests. The attachment and detachment of InPorts to and from OutPorts underlies the elimination of explicit invocation in favor of implicit invocation in the EventPorts model.

4.3 Explicit Protocol Reflection: a response to Code Pollution

A component's interface details the events in which the component is interested. Typically, the order in which those events may be received by the component is completely unconstrained. This unconstrained ordering is often problematic. For example, it would be inappropriate to allow an update lock to be set on a database record that was already locked. Thus, the events that a component is interested in receiving varies from moment to moment as a function of the mutations to its abstract state which have occurred in response to the events that it has already received. That is, the invocation ordering of a component is historically determined and hence mutable. A given component, then, exhibits an implicit time-ordered protocol.

In a traditional implementation of implicit invocation a component registers all its event interests in advance. Consequently, it is perfectly possible for that component to receive events in an order that does not respect its implicit time-ordered invocation protocol. This could lead to disastrous consequences, such as permitting an update lock on an already-locked record, as mentioned above. To respond to this problem a component's methods are typically protected against out-of-order invocation by the addition of guard code wrapped around each method's implementation to ensure that any out-of-order invocation is benign.

The problem with guard code is that it pollutes and hence complicates method implementations, thus reducing maintainability and adaptability. Furthermore, since the time ordering of protocols is implicit in the guard code scattered throughout a component’s methods, rather than explicitly recorded in a single place, it becomes difficult for the maintainers of a component to comprehend and hence respect it time-ordered collaboration protocols.

As a component becomes more sophisticated, with the addition of new data members and methods, code pollution becomes more and more problematic. This increasing need to reflect growing intra-component dependencies across methods and data members leads to fragile components unable to reflect further business process change due to the fear that “if we touch it, it might break”.

In response to the code pollution problem, we propose a direct realization of Statecharts [Harel and Politi, 1998; Harel and Politi, 1998] in a component’s implementation. Statecharts express the various abstract states in which a component (of fragment of a component) may reside over time, and connect those abstract states via transition from one to another in response to received events of interest. Thus, the current abstract state determines (via its transitions) which events are currently of interest, what action to perform in response to each event, and which abstract state to mutate to next.

In our EventPorts model [Lauder, 1999a; Lauder and Kent, 1999b] we utilize direct realization of statecharts via their association with InPorts. The Interface of an EventPorts-based Component exports one or more InPorts via which events are received. When an InPort is created, it is associated with a dynamically configurable *Statechart*. The *CurrentState* of a Statechart determines (via its Transitions) which messages the associated InPort is currently interested in receiving. If an InPort receives a message that is not associated with one of the Transitions of its Statechart’s *CurrentState*, then that message is simply ignored. If, however, that message matches the interests of a Transition, and any associated *Condition* is satisfied, then the Transition is taken, any associated *Action* (again, a first class object) is invoked, and the target State of that Transition becomes the new CurrentState.

This direct reflection of statecharts in a component's interface has the potential to eliminate code pollution, since the statechart mechanism itself ensures that an event-induced method cannot be invoked unless the time-ordered protocol inherent in the statechart is respected. The elimination of code pollution simplifies the individual methods of the component and thus eases component maintenance. Furthermore, the fact that the protocol is explicitly recorded in a single place help humans to comprehend and describe that protocol since it is no longer scattered across diverse guard code.

Since a component may have many InPorts, and each InPort is associated with a Statechart, each component may exhibit multiple orthogonal statecharts, the sum of which represents its total abstract state. The set of current states across these statecharts determines, at any given moment, the total message interests of that component. Each InPort may, therefore, be considered a porthole via which one facet of a component’s total abstract state is exported. The separation of a component’s total abstract state and state transitions into a set of orthogonal facets and associated statecharts leads to a clean separation of concerns and thus enhances component maintainability. In the legacy systems we have examined, the failure to observe such a separation of concerns into orthogonal facets has resulted in an unnecessary intertwining of fundamentally independent features. A major emphasis during the design of a replacement system, then, should be placed upon achieving such separation.

5. Summary and Further Work

In this paper we have identified six anti-patterns prevalent in legacy systems, and six desirable patterns deployable in response to those anti-patterns. It is our firm belief that the migration of a specific legacy system to a replacement system should start with an identification of the anti-patterns present in that legacy system. That is, we need to identify the problems before we propose solutions. Anti-patterns underlie a legacy system's resistance to the reflection of on-going business process change. By focusing on resolving the anti-patterns present in a particular legacy system we can ensure that we are not being seduced simply by technical wizardry, but rather are addressing the real liability inherent in the resistance to business process change. This anti-patterns driven approach, then, is our strategy to preventing legacy system migration from occurring all too literally; that is, we do not want today's replacement system to become tomorrow's legacy system. Replacement systems must be adaptive systems, and the patterns we have outlined in this paper promise to help us realize that aim.

The anti-patterns detailed in this paper, and the positive patterns that respond to them, appear to cover the majority of the technically oriented problems present in the legacy systems of our industrial collaborator. Our experience in resolving the anti-patterns via the positive patterns looks promising. It is extremely unlikely, however, that we have captured all of the anti-patterns present in legacy systems. There are almost certainly far more than the six we have highlighted here. Frequently, however, we have found ourselves thinking that we have discovered a new anti-pattern only to realize that it is a special case of the six we have already outlined. We are continuing to work with our industrial collaborator, and anticipate that new anti-patterns and patterns will emerge throughout that work.

Our on-going aim is the refinement of a generalized migration methodology, driven by the search for and response to anti-patterns present in a given legacy system. Our methodology is adaptive in that it is responsive to the addition of new anti-patterns and appropriate desirable patterns as they emerge from on-going migration efforts. In effect, then, we offer a migration process that learns from our experience, that grows with us and that captures our evolving migration expertise.

6. Acknowledgements

Our understanding of the Language Action Perspective, and Business Action Theory in particular, has benefited considerably from collaboration with Mikael Lind of the University of Boras in Sweden. Also, thanks to John Corner and Paul Gledhill for comments on earlier drafts of this paper.

This work was partially funded under EPSRC grant G16365U.

7. References

Brodie, M.L. and Stonebraker, M. (1995) *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*, Morgan Kaufman.

Ehn, P. (1988) Work-Oriented Design of Computer Artifacts, PhD Thesis. Almquist and Wiksell International, Stockholm.

Gamma, E., Helm, R., Johnson, R. and Vlissides.J. (1995a) Facade. In: Gamma, E., Helm, R., Johnson, R. and Vlissides.J., (Eds.) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. and Vlissides.J. (1995b) Observer. In: Gamma, E., Helm, R., Johnson, R. and Vlissides.J., (Eds.) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. and Vlissides.J. (1995) Adapter. In: Gamma, E., Helm, R., Johnson, R. and Vlissides.J., (Eds.) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Goldkuhl, G. (1996) Generic Business Frameworks and Action Modelling. In: *Proceedings of Language/Action Perspective '96*, Springer Verlag.

Goldkuhl, G. (1998a) The Six Phases of Business Processes - Communication and The Exchange of Value. In: *Proceedings of Twelfth Biennial ITS Conference*, Stockholm:

Goldkuhl, G., Lind, M. and Seigerroth, U. (1998b) *The Language Action Perspective on Communication Modelling: Proceedings of the Third International Workshop*, Jonkoping International Business School.

Goldkuhl, G. and Röstlinger, A. (1993) Joint Elicitation of Problems: An Important Aspect of Change Analysis. In: *Proceedings of IFIP w.g. 8.2 Working Conference on Information Systems Development: Human, Social, and Organizational Aspects*, Noordwijkerhout, Netherlands.

Harel, D. and Politi, M. (1998) *Modeling Reaction Systems with Statecharts*, McGraw-Hill.

Henning, M. and Vinoski, S. (1999) *Advanced CORBA Programming with C++*, Addison-Wesley.

Lauder, A. (1999a) *EventPorts*, PhDOOS Workshop, ECOOP 99, Lisbon.

Lauder, A. and Kent, S. (1999b) *EventPorts: Flexible Protocol Reflection*, Submitted to EDOC99.

Lauder, A. and Lind, M. (1999c) *Legacy Systems: Assets or Liabilities?*, Submitted to LAP'99, Copenhagen.

Morgan, G. (1996) Organizations as Organisms. In: *Images of Organizations*, Sage.

Riehle, D. (1996) The Event Notification Pattern - Integrating Implicit Invocation with Object-Orientation. *Theory and Practice of Object Systems* 2, 1

Schmidt, D. (1995) Reactor: An Object Behavioral Pattern for Concurrent Event Multiplexing and Event Handler Dispatching. In: Coplien, J.O. and Schmidt, D., (Eds.) *Pattern Languages of Program Design*, Addison-Wesley.

Schmidt, D. (1999) *The ADAPTIVE Communication Environment*, http://www.cs.wustl.edu/~schmidt/.

Senge, P.M. (1990) *The Fifth Discipline: The Art and Practice of the Learning Organization*, Doubleday.

SoftWired AG (1999) *iBus*, www.softwired-inc.com.

Talarian (1999) *SmartSockets*, www.talarian.com.

Tibco (1999) *Rendezvous Information Bus*, www.tibco.com.

Uschold, M. and Gruninger, M. (1996) Ontologies: Principles, Methods, and Applications. *The Knowledge Engineering Review* 11, 2

Vlissides, J. (1997) Multicast. *C++ Report* September SIGS.