JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 3, May-June 2003

Patterns for Federated Architecture

George Fernandez, Royal Melbourne Institute of Technology, Australia
Liping Zhao, UMIST, U.K.
Inji Wijegunaratne, Medibank Private, Australia

Abstract

An enterprise federated architecture intends to mirror the structure of the organisation, aiming to provide better support for both new and legacy applications within a distributed environment and facilitating data exchange between applications to support information integration. Under this architectural form, the organisation's information systems are separated out into autonomous co-operating application clusters, each connected to a message-oriented federal highway acting as the vehicle for inter-domain communication. The federated approach intends to avoid unnecessary coupling (in the distributed computing sense) by grouping highly interdependent modules and applications into domains, whilst minimising the strength of inter-domain connections. This article presents how to design a distributed federated architectural form using three architectural patterns, and shows how these three patterns are to be connected to comply with the specification of the the federated form.

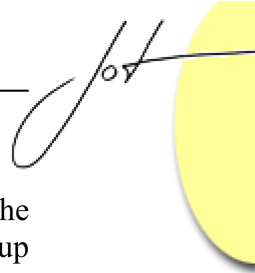
1 MOTIVATION

Let us consider this example: *SafetyNet* Insurance started as a small Australian company selling house and car insurance, but it expanded very quickly by merging with another company specialised in life insurance. The merger was to take advantage of the synergies between the different insurance products by offering their customers discounts and other advantages if they were loyal to the company by conducting all their business with *SafetyNet*. Based on the success of their sales force, the organisation expanded with branches in other capital cities in Australia. It was considered crucial that salespeople maintained at least the same level of satisfaction with their jobs, so new incentive schemes were introduced to reward the high selling branches, salespersons, etc.

The inital attempts to provide the necessary information for the new way of doing business were paper or file based. Lists of customers, sales, policy information, and other data were exchanged on paper or via files by Head Office and the different branches and groups within their branches, such as General insurance, Life Insurance, Human Resources, Finance, Actuarial Services, and Management Information Systems. Very soon a stream of problems started to emerge:

Cite this article as follows: George Fernandez, Liping Zhao and Inji Wijiegunaratne: "Design Patterns for a Federation Architecture", in *Journal of Object Technology*, vol. 2, no. 3, May-June 2003, pp. 135-149. http://www.jot.fm/issues/issue_2003_05/article4

1. *The processing of an agent's commission is slow and unreliable.* The processing of an agent's commission is housed at Head Office on a central computer that also runs another mission-critical legacy application. Each time a new insurance policy is issued by a branch, the sale information is manually loaded into a local client application, and a transaction with the central system is fired to update the commission of the employee. However, the central computer is slow, and many times the transaction is left hanging with the branch waiting for the system to respond. Often, the transaction times out, and it is necessary to kill the process and start over, frustrating the branches who get bogged down with the agent's commission, which is not even their problem. Adding to their frustration, staff members at Head Office are not very responsive when transaction logs are required back at branch level, it takes up to a week to get them, and branch staff are always under pressure to finish their reports on time. The solution of replacing the central computer is perceived as too expensive and risky by management, but they are considering it because they would like to furnish their sales force with mobile computers to do the data entry only once, and issue invoices directly when they are in the field.
2. *Identifying spending patterns to reward good customers proved to be more difficult than expected.* In the first instance, some of the applications were already interconnected by the use of files, so the same strategy was used to connect the remaining ones. Nightly processes performed the updates. However, this not only has increased significantly the number of required files, but since these files represent point-to-point connections, their format is dependent on the requirements of the two intervening applications. This situation quickly degenerated in a maintenance nightmare, and something had to be done before integrating the new systems.
3. *The IT staff members are unhappy about the complexity of the systems.* The impact of a change is always extensive, and this affects their capacity to respond rapidly to user requests. Furthermore, because of real-time interactions between functions of different modules, the system cannot function without all its components, so whenever a problem occurs, the department is under big pressure to fix the problem immediately. If a problem occurs with the invoicing system, this often means that invoicing needs to be off-line for half a working day, and this can cause a major problem in the company since the absence of a working invoicing module also affects policy data entry. This will be even more significant to SafetyNet's operations when they want to start issuing invoices in the field since this type of problems occurs quite often and they feel they will be powerless to take remedial action.
4. *There are also important autonomy and privacy considerations troubling SafetyNet.* Because of the rewards structure, each branch is required to provide insurance information to Head Office, where it is aggregated, analysed and made public. This requirement is the same for all regardless of the type of insurance, but Life Insurance groups are very concerned about this because of restrictive confidentiality clauses. Also, since the different groups are in direct competition



with each other for the rewards, they would like to have complete control of the information they are making available to the rest of the organisation, and group managers have complained about this to top management.

5. *Branch members are frustrated by the lack of autonomy.* Branch members have experienced that even when the computer system is working properly, when they require a central service their operations are severely slowed down, often grinding to a halt. More often than not they do not see the need to involve central systems on essentially local operations. They want to keep a degree of autonomy, with their own operations supported by local computer-based resources, which they believe they can manage.

Although *SafetyNet* is a hypothetical company, their problems are certainly not hypothetical. Although information technology was supposed to boost information flow within an organisation, in many cases mostly this vision has not been realised. After studying more than twenty-five companies over 2 years, Davenport *et al* [5] concluded: "Many of their efforts to create information-based organisations—or even to implement significant information management initiatives—have failed or are in the path to failure." They asserted that one of the main reasons for these failures was that information politics had not been taken into account when implementing the information infrastructure. They analysed different information models commonly found in organisations and proposed *federalism* as one of the most promising models in today's business environments, because it recognises the importance of organisational politics, and favours the use of negotiation to bring together disparate or non-cooperating parties.

2 THE FEDERATION ARCHITECTURE

Motivated by the idea of *federalism*, Wijegunaratne and Fernandez [12][13] proposed an event-driven distributed federated architectural form to enable organisation-wide distributed computing. The federated architecture intends to mirror the structure of an organisation by clustering highly interdependent modules and applications into mostly independent domains. This clustering tends to reflect the work pattern of autonomous groups in an organisation, where each domain is, in terms of administration and processing, isolated from other domains, possessing all the necessary capabilities to support its own applications. Processing and administrative isolation of domains helps reduce the complexity of the inter-domain connections. Sitting on the boundary of each domain, a software module plays the role of *domain interface* or *facilitator*, to manage and co-ordinate the information traffic of the domain with the rest of the federation. Via this interface, each application domain is connected to a message-oriented *federal highway*, to act as the vehicle for inter-domain communication and guarantee the delivery of messages (see Figure 1).

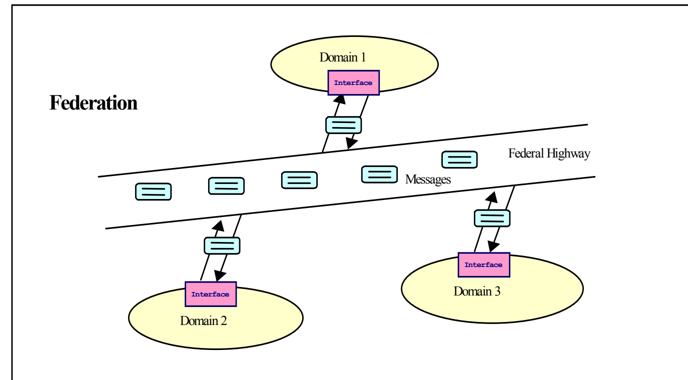


Fig.1: A federated architecture

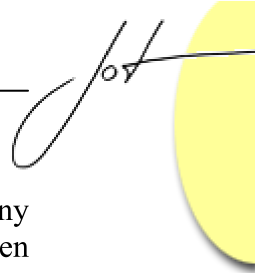
In this federated architecture, interdomain communication is event-driven, based on a publish-subscribe mechanism. The federated architecture reflects the way in which organisational departments or units interact with each other: tightly integrated software domains reflect tightly coupled people and business activities within a unit, each unit cooperating with other groups mainly to exchange information. Thus, application systems become the counterpart of the organisation, their *architecture in context* (of the organisation) [8] reflecting organisational boundaries. Such an architecture enables information exchange, and better supports distributed computing for both new and legacy applications, thus providing an infrastructure for the organisation's information integration [6].

We have identified three patterns for designing the federated architecture. These patterns characterise the structure of the federated architecture, and descibe how the architecture reflects the structure of enterprise systems. In the following sections, we present these federation patterns and describe how they are connected into a federated architecture.

3 PATTERNS FOR A FEDERATED ARCHITECTURE

The three patterns we have identified are: THE FEDERATION, DEPENDENCY SEPARATION and INTERFACE CONNECTION. THE FEDERATION describes the overall structure of the federated architecture. Process dependencies are handled by DEPENDENCY SEPARATION. Information exchanged between application domains is facilitated by INTERFACE CONNECTION.

Our notion of pattern is in line with that of Alexander [1]. We believe that patterns are connected. The connection of patterns and the sequence in which patterns are applied give rise to pattern languages. Although there are only three patterns in our design, they are connected into a pattern language which is used to describe and design the federated architecture.

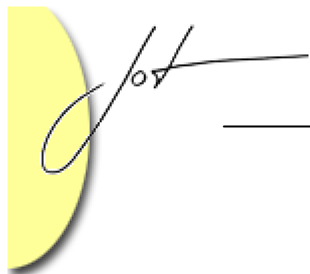


Since the federated architecture intends to reflect the structure of many organisations, patterns in this language are naturally connected to some of the Coplien organisational patterns [3]. These patterns attempt to parallel the human and computer system aspects of the organisation. As it happens in organisations, we argue that our patterns can be connected to other patterns in a web of patterns.

In describing our patterns, we have used the Alexander pattern form ([1], pp.x-xi), which has the following components:

> *First, there is a picture, which shows an archetypal example of the pattern. (Our note: This has been omitted for brevity)*
>
> *Second, after the picture, each pattern has an introductory paragraph, which sets the context for the pattern, by explaining how it helps to complete certain larger patterns.*
>
> *Then there are three diamonds to mark the beginning of the problem.*
>
> *After the diamonds there is a headline, in bold type. This headline gives the essence of the problem in one or two sentences.*
>
> *After the headline comes the body of the problem. This is the longest section …*
>
> *Then, again in bold type, like the headline, is the solution -- the heart of the pattern -- … This solution is always stated in the form of an instruction -- so that you know exactly what you need to do, to build the pattern.*
>
> *Then, after the solution, there is a diagram, which shows the solution in the form of a diagram…*
>
> *After the diagram, another three diamonds, to show that the main body of the pattern is finished.*
>
> *And finally, after the diamonds there is a paragraph which ties the pattern to all those smaller patterns in the language, which are needed to complete this pattern, to embellish it, to fill it out.*

In addition, at the end of each pattern, we have used function notes to discuss other similar patterns. The function notes are similar to the **Known Uses** or **Related Patterns** in [7].



4 PATTERN 1: THE FEDERATION

To function better, an organisation wants to make possible information sharing and integration by developing organisation-wide distributed computing.

Information sharing and integration within an organisation is facilitated by a systems architecture that supports consistent and reliable information flow between different application systems.

To build such architecture, it is necessary to understand how information is produced, who owns it, and how computer applications are placed to support organisational work. It is typical of an organisation to be divided into groups with considerable autonomy and decision-making power such as departments, divisions or cost centres. These groups usually have their own computer systems, autonomous and independent of systems of another group, to support their daily work. Information sharing and integration require that these groups and their systems expose their information for sharing.

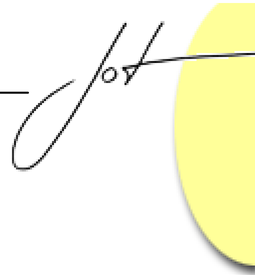
However, sharing of computer information can be problematic, since it implies the creation of inter-application connections. As a result, as demonstrated by the *SafetyNet* example, the number and diversity of application to application connections becomes unmanageable, and the autonomy and independence of applications can be compromised. Sharing also entails a security problem, as it can be a threat to data privacy and integrity.

The aim should be to preserve applications independence and data integrity in an organisation, and at the same time enabling them to gain access to each other's information in a controlled manner [12]. As seen in the *SafetyNet* example, serious problems arise when interconnections do not reflect organisational structures: the systems' scope of authority, command, and control do not reflect the groups' authority, command, and control.

More than three decades ago, Melvin Conway [2] suggested that the structure of the system should mirror the structure of the organisation that designed it. This has since become known as Conway's Law, and has been supported by many empirical studies [3][10].

Coplien [4] explained why the homomorphism between the organisation and system architecture is important:

> *Architecture is not so much about the software, but about the people who write the software. The core principles of architecture, such as coupling and cohesion, aren't about the code. The code doesn't care about how cohesive or decoupled it is; if anything, tightly coupled software lacks some of the performance snags found in more modular systems. But people do care about*



> *their coupling to other team members. The structure of the organization that builds the software is homomorphic to the structure of the software; it's not always clear which is cause and which is the effect, so we'd better get both right.*
>
> *Architecture, like any system discipline, is about relationships between system parts, and according to Conway, between people.*

Davenport *et al.* [5] observed that a model that recognises organisational politics and admits conflict would better reflect reality than a model of unstinting cooperation and unfettered exchange of information. Thus, they suggested federalism as a preferred archetype, where potentially competing or non-cooperating parties are brought together by negotiation.

Therefore,

Identify closely related (people and suporting systems) activities in an organisation, along geographical, process, or functional lines. Divide thus the organisation into clusters (domains) with their own administrative and control mechanisms, and supporting systems. The internals of a domain are to be opaque to other domains. Let domains communicate exclusively via messages to be placed on a federal organisation-wide information highway guaranteeing message delivery. Data flow is only by a domain publishing, and other domains subscribing to, specified agreed upon information. A domain itself can consist of a federation; therefore, in this sense, this federation pattern is recursive.

Inter-domain process dependencies must be resolved correctly to ensure effective information flow. THE FEDERATION needs the support from DEPENDENCY SEPARATION.

Function notes:

Alexander observed the importance of the congruence between physical spaces and social spaces. He proposed to make the physical structure of a building conform to the structure of social spaces – STRUCTURE FOLLOWS SOCIAL SPACES ([1], pp. 940-945). THE FEDERATION intends to reflect the information 'working space' of organisations.

THE FEDERATION architectural style is a popular approach for agent systems [9]. In a multi-agent system, agents communicate with each other to exchange information and services. This can include *direct communication*, in which agents handle their own co-ordination; *assisted co-ordination,* in which agents rely on special co-ordination systems; and *federated systems,* in which agents communicate directly to their local *facilitators*, which in turn communicate with agents on different locations. The structure of an agent-based federated system is illustrated in Figure 2, where agents are located in

three different computers, A, B, and C. Each computer has a local facilitator as a contact point for the agents within the computer, and for other facilitators in other computers.

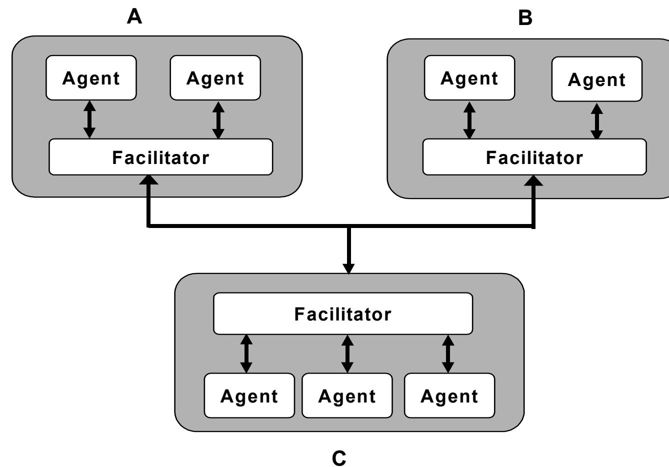


Fig. 2: A federated agent communication system (after [9])

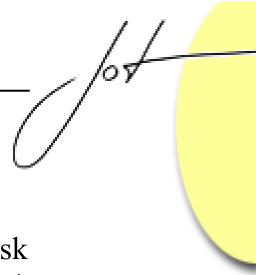
5 PATTERN 2: DEPENDENCY SEPARATION

This pattern enables THE FEDERATION by resolving dependencies between applications within domains.

Grouping applications into clusters or domains reduces the number of required connections between applications; domains exchange information only in terms of messages. Within a domain, all inter-application dependencies and connections are allowed.

Wijegunaratne and Fernandez [12] noted that dependencies between applications stemmed from the way people activities were conducted by an organisation. Dependencies in software ought to reflect an organisation's form of business activities, they should be the result of the way the organisation chooses to work and, therefore, the nature of the dependencies must be made explicit in the architecture. They have identified the following types of software dependency stemming from people's activities:

1. *Processing Dependency*. An application module requires some work to be carried out remotely by other application modules in order to complete its own processing. Processing dependency may fall into two categories:
   – *Simple Processing Dependency* where an application module needs another (probably remote) application module to perform some task before it can proceed or complete processing. This is a simple dependency.



- *Transactional Dependency* where an application module requires several application modules on different, probably remote, sites to carry out some task before it can progress. The operations must be carried out in an 'all or nothing' fashion, such as in a banking transaction.

2. *Informational Dependency*. An application module needs to convey some information to one or more remote application modules as a consequence of some event within its jurisdiction.

Wijegunaratne and Fernandez suggested that these dependencies should be ranked according to the coupling (in the distributed computing sense [12][14]) they introduce in the system. The highest coupling corresponds to transactional dependencies, followed by simple processing dependencies, while informational dependencies represent the loosest coupling between applications. Problems may arise when dependencies with lower coupling (a requirement matter) are implemented as an interaction (a design issue) with higher coupling, such as when an informational dependency is unnecessarily implemented as a transaction, because they reduce application independence. This can be seen in the *SafetyNet* systems (Problems 1, 3 and 5 above), when the agent's commission (an informational dependency) is implemented as a transaction, because then local system's autonomy is unnecessarily curtailed by central processing, and local administrators resent their systems performance being degraded by slow and unreliable remote processing.

Two modules linked by informational dependencies can be connected by messages apprpriately triggered by local events. Further more, since there is no processing dependency involved, the communication may be asynchronous—the recipient does not even need to be available for the sender to hand the message over to the delivery mechanism—thereby enhancing application independence. Processing dependencies introduce tighter coupling than informational ones and, therefore, they impinge on domain independence; this goes against the natural tendency to remain autonomous of the groups that the domains represent. Wijegunaratne and Fernandez concluded that it would be possible to achieve the domains' required *processing* and *administrative isolations* if only informational dependencies were allowed between domains.

This separation is not always possible. Sometimes, a processing (simple or transactional) dependency exists between modules. If this is so, the modules must belong to the same domain. However, if the domain boundaries have been determined following natural orgnisational fissures, it is highly unlikely that a processing dependency cannot be reformulated as an informational one. Therefore, domains must be demarcated such that processing dependencies between domains are able to be restructured into informational dependencies. Although within a domain processing and informational dependencies may co-exist, it is also desirable to restructure dependencies the same way.

Therefore,

Identify processing and informational dependencies according to the organisation's rules, policies and business activities. Treat the dependencies in the following order.

1. **Translate processing (transactional and simple) dependencies into informational dependencies to ensure that no processing dependencies exist between applications belonging to different domains.**
2. **Within each domain, while both processing and informational dependencies can co-exist, wherever possible: (a) translate processing dependencies into informational dependencies and (b) translate transactional dependencies into simple processing or informational dependencies.**

Application dependencies within THE FEDERATION architecture have now been dealt with by DEPENDENCY SEPARATION. It is the time to provide support for domains information exchange – INTERFACE CONNECTION.

6 PATTERN 3: INTERFACE CONNECTION

THE FEDERATION and DEPENDENCY SEPARATION have resolved inter-domain and inter-application dependencies. This pattern connects domains for organisation-wide information exchange and integration.

Federal, organisation-wide information exchange requires interoperation between the domains. To minimise individual domain effort, a domain-based communication mechanism should be established to make possible information flow.

Given `n` applications there can be up to `n*(n-1)/2` application-to-application links, as shown in Figure 3. Point to point application interfaces proliferate rapidly as `n` increases.

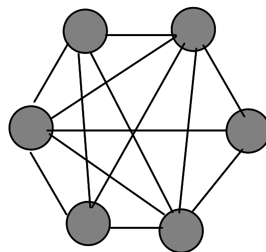
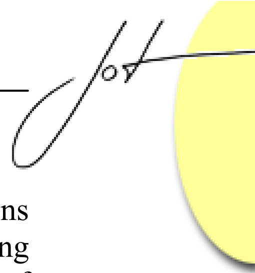


Fig.3: N applications with N(N-1)/2 inter-communication channels (After [11])



Too many inter-application links threaten application integrity and make applications difficult to maintain and adapt. An error, an abnormal condition, or a "hanging running module" occurring at run-time in an application can propagate to a large number of applications; a change to an application will also affect a larger number of other tightly coupled applications. Meyer [11] proposed a solution called *Few Interfaces* to reduce the number of communication channels, in which there is a centralised "boss" application responsible for inter-application communication. With this solution, inter-application links have been reduced to a minimum number of `n-1`.

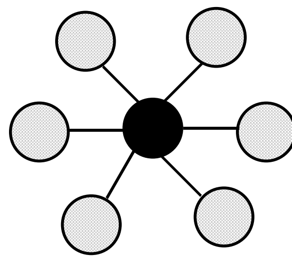


Fig. 4: N applications with (N-1)/2 inter-communication channels (After [11])

But this solution simply redirects all the dependencies into one central module; it doesn't solve the problem of effective communication. It reduces the coupling, but it doesn't facilitate information flow.

Given that only informational dependencies exist between domains, a domain never requires any processing from a remote application in another domain. Domains are therefore only producers and consumers of corporate information. A producer domain is only required to broadcast—"send and forget", "tip over the wall"—the required information, ignoring which (consumer) domains are interested in that piece of information.

Every application in a domain interacting with every other application in other domains would create a high number of interconnections between applications, as discussed in *SafetyNet*'s file-based transfer. Such interconnections are inconsistent and need to be handcrafted every time a new one or an update are required. In addition, application-to-application communication would immensely increase the cost involved in broadcasting information, since there is no standard format but only pair-wise agreements. This situation is similar to the problem of communication between agents [9] and software development teams [3].

In the agent community, this situation is dealt with by delegating communication between agents to their facilitators [9]. Agents are grouped according to their locations and functions, much the same way as domains in the federation. Each agent group has a facilitator.

In software development, each project team selects a public character as a gatekeeper (see the GATEKEEPER pattern in [3]), whose responsibility is to disseminate information from outside the project to project members and translating it into terms relevant to the project. This gatekeeping mechanism should also be implemented for the federation to channel all connections between domains. It services a domain by placing messages on the highway for consumption, and picking up from the highway messages intended for the domain to deliver them to the relevant applications.

Therefore,

A domain interface agent is in charge of picking up messages from, and placing messages on, the federal communication infrastructure. This agent plays the following roles for each domain in the federation.

1. ***Message delivery service*. To guarantee the delivery of the requested information to consumer applications within a domain.**
2. ***Domain information repository management*. to store, manage and use format, security and other federal information.**
3. ***Publish/subscribe directory service*. To manage publisher/subscriber information related to the domain.**

Function notes:

The INTERFACE CONNECTION brings the independent domains together to work co-operatively. In this respect it has the similar importance as CONNECTION TO THE EARTH ([1], pp.786-788), which connects the building to the earth.

It is worth noting here the close correspondence between the three federation patterns and the MEDIATOR pattern [7]. When the federation patterns are applied to object-oriented programming, their collective effort is the MEDIATOR pattern (see Figure 5). In effect, the MEDIATOR pattern contains the three federation patterns. THE FEDERATION divides colleagues according to their behaviour; DEPENDENCY SEPARATION is achieved by the colleague inheritance hierarchy. Finally, the mediator inheritance hierarchy serves the purpose of INTERFACE CONNECTION.

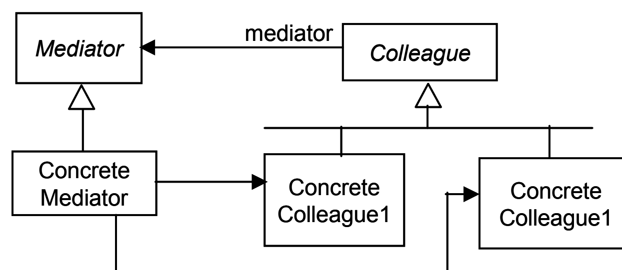
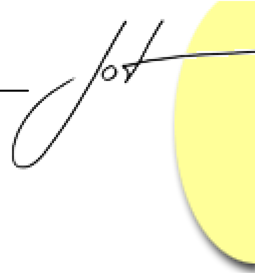


Fig.5: The Mediator pattern (after [7])



7 CONCLUDING REMARKS

The federation patterns presented above will have a significant impact on *SafetyNet*'s operations and can solve the problems presented in the Motivation section:

- Operations unnecessarily implemented as transactions, such as agents' commissions at *SafetyNet*, introduce complexity not existing in the organisation workflow since the only requirement is for the information to be reliably relied to the target system. The commission can be updated by an appropriate message when a new policy is issued, so decoupling the processing of branches and Head Office (Addressing *SafetyNet*'s Problem 1).
- A reliable delivery mechanism with agreed interfaces makes possible information sharing but avoids point-to-point connections, reducing the number of structures and formats required (Addressing Problem 2).
- Systems maintenance is simplified since there is no processing between domains, and the scope of (processing) changes is contained within a domain. The independence of domains ensures that poorly performing applications in one domain don't impinge on the performance of applications in other domains. Since modules in different domains don't even need to be simultaneously available, bringing a domain module off-line has no impact on other domains. (Addressing Problems 3 and 5).
- Each domain retains control of the information it publishes (Addressing Problem 4).
- An event such as invoices issued in the field can be sent as asynchronous messages when convenient (even batched), since there is not need for synchronisation with a receiving module, not even for it to be available (Addressing Prolem 3).
- Domains retain complete control of local processing (Addressing Problem 5).

There is also another approach to federation patterns. This would describe the federation from the point of view of the implementation rather than the requirement, and include patterns such as “The Domain”, “The Message Interface Agent”, etc. This is a matter of ongoing research.

ACKNOWLEDGEMENTS

This article is based on an early paper presented at KoalaPLoP’2000. We wish to thank all the workshop participants for their valuable comments and suggetions. In particular, we are indebit to our KoalaPLoP shepherd, Neil Harrison for his valuable help with the early version of this paper.

REFERENCES

[1] C. Alexander *et al*. *A Pattern Language*. Oxford University Press, New York, 1977.

[2] M. Conway. "How Do Committees Invent?" *Datamation*, Vol.14, No.4 Apr. 1968, pp.28-31.

[3] J. O. Coplien. "A Generative Development – Process Pattern Language." In *Pattern Languages of Program Design*. J .O. Coplien and D. C. Schmid (eds.), Addison Wesley, 1996.

[4] J. O. Coplien. "Reevaluating the Architectural Metaphor: Toward Piecemeal Growth." Guest Editor's Introduction. In *IEEE Software*, Sept/Oct, 1999, Vol.16(5).

[5] T. H. Davenport, R. G. Eccles, and L. Prusak. "Information Politics." *Sloan Management Review*, Fall, 1992.

[6] G. Fernandez and I. Wijegunaratne. "A Cooperative Approach to Distributed Applications Engineering." Proc. Asian'96 WORKSHOP, Coordination Technology for Collaborative Applications: Organisations, Processes and Agents, Dec 5, 1996, pp 39-48.

[7] E. Gamma, *et al*. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, ©1996.

[8] D. Garlan D., R. Allen, J. Ockerbloom (1995): "Architectural Mismatch: Why Reuse Is So Hard", *IEEE Software*, Nov 1995, pp 17-26.

[9] M. R. Genesereth and S. P. Ketchpel. "Software Agents." In *Communications of The ACM*, 37(7), July, 1994.

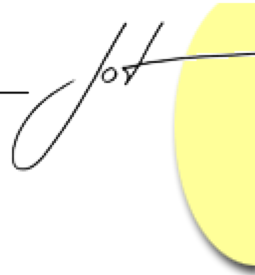
[10] J. Herbsleb and R. E. Grinter. "Architectures, Coordination, and Distance: Conway's Law and Beyond." In *IEEE Software*, Sept/Oct, 1999, Vol.16(5).

[11] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall. New Jersey, 1988.

[12] I. Wijegunaratne and G. Fernandez. *Distributed Applications Engineering*. Springer, London, 1998.

[13] I. Wijegunaratne, G. Fernandez and J. Valtoudis. "A Federated Architecture for Enterprise Data Integration." *Proceedings of the Australian Software Engineering Conference*, Canberra, Australia, April 2000.

[14] E. Yourdon and L. Constantine. *Structured Design*. Prentice Hall. New Jersey, 1978.



About the authors

George Fernandez is an Associate Professor (Information Systems) at the School of Computer Science and Information Technology at RMIT. George has more than 25 years of experience in Computing and Information Systems, working in academia, private industry and government organisations, both in Australia and Argentina. His research interests include Federated Architectures for the Enterprise, Distributed Computing and Data Integration, and Information Systems Security. George may be contacted at gfernandez@rmit.edu.au.

Liping Zhao is a lecturer at the Department of Computation, UMIST, U.K. Her research interests include object-oriented design and design patterns. She can be reached at liping@co.umist.ac.uk

Inji Wijegunaratne received his PhD from London University. He has over 20 years experience in the IT industry. Since the mid 90's Inji has been working in the IT architecture area, where he has implemented these patterns in real situations. He is currently Manager, IT Architecture at Medibank Private in Australia. Inji may be contacted at inji_wijegunaratne@medibank.com.au.