Supporting the Evolution of Service Oriented Web Applications using Design Patterns

Manolis Tzagarakis
Computer Technology Institute
26500 Rion
Greece
+30 2610 960482
tzagara@cti.gr

Michalis Vaitis
University of the Aegean
University Hill, GR-811 00 Mytilene
Greece
+30 22510 36433
vaitis@aegean.gr

Nikos Karousos
University of Patras
26500 Rion
Greece
+30 2610 960482
karousos@cti.gr

ABSTRACT

Web applications make increasingly use of services that are provided by external information systems to deliver advanced functionalities to end users. However, many issues regarding how these services are integrated into web applications and how service oriented web applications evolve, are reengineered and refactored are still addressed in an ad hoc manner. In this paper, we present how design patterns can lessen the efforts required to integrate hypermedia services into web applications. In particular we present how evolution and maintenance issues are addressed within Callimachus, a CB-OHS that web applications need to integrate in order to provide hypertext functionality to end users.

.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]:

General Terms

.

Keywords

Web application, service oriented architectures, hypertext.

1. INTRODUCTION

The term "web application" characterizes a particular class of applications that make use of internet technology to deliver content and services such as HTTP, HTML, XML and Web Services [1]. One particular class of Web applications deals with integrating information systems into web applications.

Web applications are still developed in an ad hoc manner, resulting in applications that fail to fulfill several importand requirements including:

1. User needs, meaning that the web application is not what the user wanted
2. Easy maintenace and evolution
3. Long useful life
4. Performance and security.

As already pointed out in [13]

"*Web systems that are kept running via continual stream of Patches or upgrades developed without systematic approaches*"

The problems are even more complicated, when web applications are built upon service oriented architectures (SOA) that differ greatly fom traditional client server architectures. SOA exhibit great flexibility with respect to services and require new approaches to service integration. Within the hypermedia field, Component-Based Hypermedia Systems (CB-OHS)[15] have emerged, consisting of an underlying set of infrastructure services that support the development and operation of an open set of components (called structure servers), providing structure services for specific application domains. The theoretical and practical aspects of this promotion of structure from implicit relationship among data-items to a first-class entity constitute the subject of the field of structural computing [5]. Attempts to integrate services provided by CB-OHS with web applications are already underway [14].

CB-OHS are among the forerunners of a trend for service-oriented computing (SOC) [8]; the computing paradigm that utilizes services as fundamental elements for developing applications [2] and relies on a layered SOA. A SOA combines the ability to invoke remote objects and functions (called "services") with tools for dynamic service discovery, placing emphasis on interoperability issues [3]. As both hypermedia applications and the class of web applications categorized as informational [1] are content-intensive, the employment of structure services (following the SOC paradigm) would improve efficiency and convenience [9].

Unfortunately, today's developers of hypermedia and web applications face various problems when attempting to integrate services offered by CB-OHS into web applications. This is in particular true when considering evolution and maintenance issues. Currently, such concerns are addressed by developing structure services from scratch [4] redesigning appropriately the provided services. We argue that one of the reasons for this situation is the lack of both an adequate software engineering framework for CB-OHS construction, integration, and maintenance and the appropriate tools to support it. This results in ad-hoc integration methodologies which produce systems missing certain essential characteristics including difficulty to evolve and maintain.

In this paper, we present how design patterns can lessen the efforts required to integrate hypermedia services provided by service oriented systems into web applications. In particular we present how evolution and maintenance issues are addressed within Callimachus, a CB-OHS that web applications need to integrate in order to provide hypertext functionality to end users.

The paper is structured as follows: first we outline aspsects of SOA that makes integration into web applications difficult and error prone. We then present Callimachus and how the services provided are integrated into web applications. Next, we present and analyse the design patterns that are used to address evolution and maintenance concerns. Finally, future work concludes the paper.

2. Service Oriented Architectures (SOA)

Traditionally, hypermedia systems have been built according to client server (or point-to-point) architectures that provided an adequate framework for bringing hypertext functionality to web applications. However, the design and development of these hypermedia systems were based on assumptions that reflect the architecture upon which they were developed. Moving hypermedia systems to service oriented architecture requires these assumptions to be re-examined and adjusted. This is because service oriented archtectures differ greatly from client server architectures. Table 1 summarizes the main differences between service oriented and client server architectures.

In service oriented architectures, bindings to services (i.e. references to operations provided by services) are established dynamically and during runtime which is completely incompatible with client server based hypermedia systems where such binding of clients to services happen very early in the development process (in particular during design or compile time). At run time, changing bindings is impossible.

Table 1. Comparison of Client Server vs Service-Oriented Architectures

Client Server Architectures	**Service Oriented Architectures**
Early binding (compile/development time)	Late binding (run time)
Domestic (evolve smoothly and planned)	Feral (evolve abrupt and uncontrolled)
Location dependent	Location independent and transparent
Single interface (protocol)	Set of interfaces (protocols)
Development oriented	Integration oriented
Tightly coupled	Loosely coupled
Monolithic	Composable
Stable	Unstable due to ad hoc nature

While client server architecture evolves in a controlled and disciplined fashion, service oriented evolves in a rather feral way. This is mainly due to the autonomous nature of services that implies an autonomous evolution path as well. As a result client-side bindings to hypermedia services can easily be invalidated. In addition, it is evident that while client server architectures exhibit location dependence thus forbidding changes in location information (e.g. in terms of host and port) service oriented architectures are location independent making conventional clients unable to operate in such an environment. With respect to the supported interfaces, in client servers systems only a small, bound number of interfaces are supported whereas in service oriented systems an unbound number of interfaces exists. Thus while in client server systems it is enough for all software entities (e.g. client application) to be reactive when considering interfaces to hypermedia services, in service oriented architectures all software entities need to be proactive. Furthermore, client server systems are tightly coupled systems, meaning that design changes in the service are followed by design changes on the client side. This is not the case in service oriented characterizing this architecture as loosely coupled. Finally, while in client server systems the main task during development is to extend the client and the server respectively, in service oriented architectures the main task of a developer is to integrate services.

From the above discussion it is clear that service oriented architectures represent an environment where all software entities need to exhibit flexibility, autonomy, and adaptability in order to function correctly and take advantage of the plethora of services that are presented. Within such an agile environment, web developers require new tools and infrastructures that will enable smooth evolution as well as seamless integration of the provided services into web applications.

2.1 The Callimachus Component Based Hypermedia System.

Callimachus is an open hypermedia system [6, 7] that aims at providing hypertext functionality to an open set of applications. It provides support for wide range of domain specific abstractions thus addressing a broad range of hypertext domains [5]. Such domains include *navigation*, allowing the interlinking of information and *taxonomic reasoning* to develop for example directory services on the world wide web [14].

Callimachus follows a component-based architecture as depicted in figure 1. Each component provides a number of services through which clients can request domain specific hypertext functionality. Its primary architectural elements are client applications, structure servers and infrastructure. Client applications can be either native or third-party applications, such the MS Office Suite and Emacs, or even web servers and entire web applications. Client applications (clients for short) request services from structure servers using a well defined protocol. Structure servers provide the domain specific abstractions of a particular hypermedia domain by offering a consistent set of services. The infrastructure provides services across hypermedia domains such as storage, naming and notification.

The on-the-wire messages sent between clients and structure servers are encoded using XML and transferred using HTTP tunneling. The adoption of this technique has been imposed mainly by the need to overcome the access restrictions to non WWW services enforced by firewalls. HTTP is used as a transport protocol to tunnel client requests. The Content-Type parameter specifies the protocol that is being used.

All client-side aspects of the protocol come in the form of a library that implements an API. Different structure servers require different protocols to communicate with client applications. The construction of the client-side API takes place during the development of the structure server. In Callumachus, all structure servers have the form of a TCP/IP daemon listening on a specific port for incomming requsts. Each structure server can serve concurrently many clients that can be of different types (e.g. web application, Emacs etc).

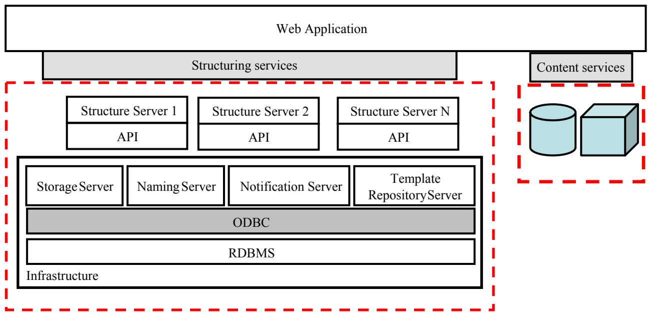

Figure 1: The conceptual architecture of Callimachus and how it is integrated with web applications

Being clients, web applications request structuring services from Callimachus and content services form other information systems. For example, in case the web application provides directory services, it invokes structuring services such as openCategory or getPathOfCategory from Callimachus, and it resolves the returned content identifiers using the content services. At the web application layer, the outcome of both service invokations are merged and transformed to the appropriate format (e.g. html or XML). The result is then sent back to be displayed to end users.

The development of structure servers and the integration mechanisms (i.e. APIs) follows an evolutionary rapid prototyping approach with short iterations and many releases. This means that there is a constant evolution of services with which the entire framework has to cope with.

Design and development is split into tasks, each one dealing with a particular aspect of the structure server. Three main tasks are carried out, each producing a prototype subsystem. The integration of developed subsystems results in a working structure server. The specification, design and implementation of each subsystem does not follow a particular process model, because of their tightly coupled nature and their "small" size as software artifacts. These tasks are described more detailed below.

Server shell development: During server shell development, the structure server's interface is built. In this task, the emphasis is on the design of the exact procedure the structure services are invoked. More precicely, all aspects of the structure server when viewed as the receivers of client requests are addressed. Such aspects include listening to, parsing and validating incoming requests, as well as preparing and passing these requests to the domain model for execution.

Domain model development: During this task, the syntactic and behavioral aspects of the domain-specific abstractions (including their relationships) are designed and developed. The syntactic and behavioral specifications originate from the scenario and are defined in terms of the Callimachus Abstract Structural Element.

Integrator development: The aim of this task is the development of the necessary software modules that will enable integration of clients with the structure server. These modules come in the form of a client-side API. Specifically, a wrapper container and a communicator are developed [52] so that client applications are able to request structure services.

The prototyping phase starts with the development of an initial domain model prototype. Consequently, the server shell and the integrator prototypes are developed. After an initial cycle, each prototype is refined by constantly iterating through the tasks until an acceptable structure server prototype has been completed. The prototype structure server is tested by end-users aiming to assess its accordance to the scenario.

These challenges include both non-functional and functional aspects of structure servers:

Incremental service (and operation) formalization: During prototyping, the set of the provided services (and operations that clients can request) is initially unknown, with their name, behavior and parameters slowly emerging, as prototypes become available for testing. By having services emerging and evolving while development is progressing, the emphasis is on ways to easily integrate new or modify existing services, without requiring changes in functionally unrelated modules of the structure server (which cause major concerns to developers). In particular, the goal here is to achieve localization of the effects during the evolution of services.

Smooth evolution of protocol implementations: Although the design of multi-protocol support ensures easy integration of new protocols developed entirely from scratch, it does not address evolution of existing protocols. During protocol evolution, new methods might be added to existing protocol implementations; existing methods might change their signature or might even be associated with different operations at the domain model layer. Such tasks need to be carried out quickly to ensure short iteration cycles.

3. Design Patterns

Within the Callimachus project, design patterns [11] have been proven a valuable mechanism to support smooth evolution of hypermedia services and their seamless integration with Web applications.

In particular, design patterns are utilized to address changes at the hypermedia services layer due to new web application needs as well as changes at the web applications layer due to changes at hypermedia services. Consequently, two types of design patterns can be identified: patterns that address concerns at the hypermedia services layer and patterns that address concerns at the web application layer.

Next, for each layer, we briefly present the design patterns used. Although the design patterns discussed are already well known, the focus is mainly on what benefits can be gained when using them in service oriented environments.

3.1 Design Patterns at the Hypermedia Service Layer

3.1.1 Protocol Handlers

Everytime a connection with a client is establised, all received requests for structure services need to be parsed in order to be checked for validity and prepared for execution. Validity checking includes the examination of the conformance of the requesting message to the syntax of the domain protocol specifications, as well as to the semantics of the domain model functions (i.e., the indicated operations along with the type of parameters supplied). Preparing a request for execution refers to the necessary actions dealing with determining the appropriate operation in the domain model that has to be executed. Such tasks are the responsibility of the protocol handler [12]. Since different structure servers require different protocols, development of protocol handler is performed every time a new structure server is developed. The situation gets more perplexed when considering that the same structure server can be accessed using different protocols meaning that the same structure server needs to provide support for a number of protocols that need to be activated at runtime. The question thus is how to make the same set of operations provided by structure servers available through different protocols.

To achieve smooth evolution of protocol issues within structure servers, parsing of incomming requests must be decoupled from invocation of the operation that requests designate. For this reason, the strategy design pattern is used [11]. This permits also, the parsing algorithm to vary according to the incomming request.

How the strategy design pattern is utilized is depicted in figure 2. Within each structure server, the ServerContext class deals with all low level aspects of receiving a request from the TCP/IP socket, as well as parsing the HTTP headers of the tunneled request. The class also maintains a reference to an instantiation of the HypertextProtocol, an abstract class that is used to parse the received request and supports only the public virtual methods `Parse` and `Clone`. While the `Parse` method encapsulates the suitable algorithm for parsing and preparing incoming requests, the `Clone` method returns a copy of the HypertextProtocol instance, used in the context of the prototype design pattern. All protocols supported by a particular structure server, are derived from the HypertextProtocol class. Every derived class (that constitutes a protocol handler) implements the method parse, where the appropriate code for parsing, validating and preparing the request is placed by the developer. The appropriate protocol is determined and instantiated during runtime based on HTTP's Content-Type parameter. For this task, the prototype design pattern is utilized, determining how the appropriate available protocol implementations are declared and instantiated during runtime. The hypertext protocol factory is part of the ServerContext class and is instantiated during initialization of the structure server. There is exactly one hypertextProtocolFactory for every structure server.

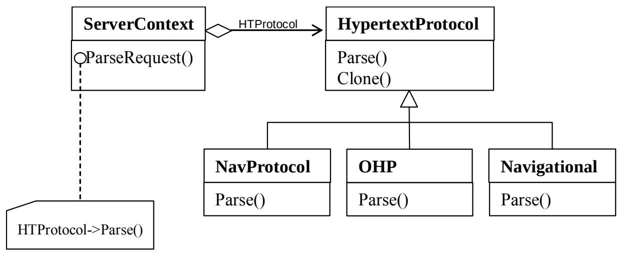

Figure 2 : Protocol Handler

Adding support for new protocols is fairly trivial, allowing developers to focus only on parsing and preparing without spending time about how to integrate the new protocol into the structure server. During design time, developers have to create a class that resembles their protocol implementation (derived from the `HypertextProtocol` class) and to provide the implementation for the `Parse` method. Furthermore, they have to register the new class in the factory's `registerProtocol` method that takes place in the factory's constructor. During runtime, correct deployment of the new protocol handler is ensured by the prototype design pattern [11], since the mechanism of how to determine which class to instantiate is independent of protocol handlers.

3.1.2 Service Execution

Different web applications may require different set of operations from the same structure server. For example some web application providing directory services might require complex editing of entire subtrees such as deleting directories or moving and copying them to different locations while others don't. Moreover, for all available operations, undo, redo, logging and queuing options should be available. The question here is how to systematically extend the available operations (and thus services). The goal is to provide domain specific operations in a plug-and-play fashion. To support such development tasks, the invocation of an operation needs to be seperated from its execution. Within Callimachus, this is achieved using a variation the active object and command processor design patterns [10].

In the design pattern of figure 3, all client requests (denoting operations, such as openNode, traverseLink in case of a navigational structure server and deleteDirectory in case of a taxonomic structure server) are instantiated as separate objects. There exists one class for each operation available to clients, elevating operations to first class entities, thus allowing them to be stored, scheduled and even undone. Such treatment of operations also allows the support of transactions. All available operations are derived from the DomainOperation class, an abstract class with two methods: `Execute` and `Undo` (implemented by the concrete derived classes). The Execute method of each concrete class executes the operation by calling the appropriate method of the class HMDomain that represents the interface to the domain model subsystem. For example, the openNode class would call the openNode method of class HMDomain.

The appropriate concrete operation instances are created by the HypertextProtocol class, after having parsed and validated incoming client requests. The HypertextProtocol class decides which operation to instantiate in order to be flexible with respect to which method of HMDomain class to invoke. There might be cases where a matching method might not be available in the HMDomain class, so an equivalent method (or set of methods) in that class should be invoked. For example, a getNode operation (that would be modeled as a separate class) has to invoke the available openNode method (i.e., an equivalent method) of the HMDomain class, when a getNode method is not available. Such choice is conveniently done in the HypertextProtocol class after parsing and before the execution phase of client requests.

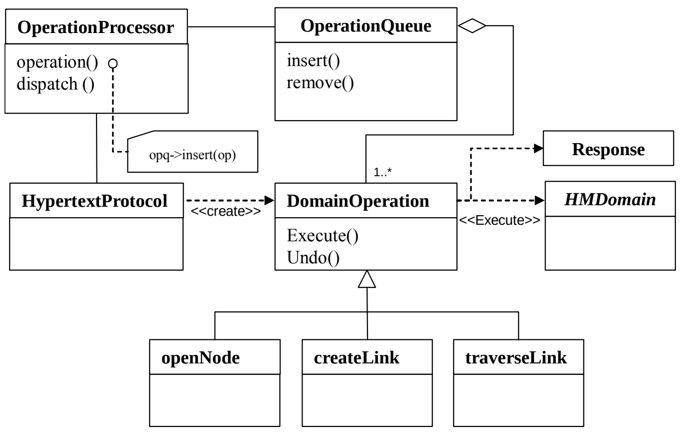

Figure 3: Service execution

The HypertextProtocol class enqueues all operation instances by calling the Operation method of the OperationProcessor class. There is exactly one OperationProcessor instance for every structure server. Thus, an OperationProcessor constitutes a singleton [12]. The OperationProcessor class maintains the operation objects in the OperationQueue, and schedules their execution. The OperationQueue class may arrange the operations by priority and decide which operation is ready to be executed by calling the operation's canExecute method. Operations are dequeued and executed concurrently by calling the appropriate methods of the HMDomain class. Each operation executes in a separate thread of control. The output of each operation is available through a specific class (see Response class in Fig. 3) that is used to send replies back to clients.

During structure server evolution, developers can systematically approach the problem of constant change in the domain operations, in the protocol specifications and in their bridging. New operations can be added during design time by extending the DomainOperation class and delegating execution to the appropriate domain specific interface method. Since identification and invocation of the operation are provided by the framework at run-time, developers can focus only on semantic aspects of the operations. In addition, the framework provides the foundation for supporting a number of advanced (but necessary) capabilities, such as the undo/redo operations, as well as transaction management for all structure servers in a uniform manner, thereby reducing maintenance efforts.

3.2 Design Patterns at the Web Application Layer

While the previous sections presented design patterns that facilitate the evolution of structure server when new web applications requirements emerge, the following design patterns address concerns at the web application layer and in particular attempt to address issues that deal with hypermedia service invocation.

With respect to invocation, the patterns aim at providing mechanisms to achieve the following:

1. provide a single point from which requests to the hypermedia services originate.
2. Offer templating mechanism for re-occuring invocation schemes.

3.2.1 Single Invocation Point: Dispatching requests

Everytime developers need to issue requests from the web application layer to Callimachus they place code (e.g. that uses the API for accessing hypermedia services) in different web application modules. Such approach to hypermedia service provision results in code that is unstructured and thus unmaintainable. The question here is how can be hypermedia service invocations be systematically integrated into web applications reducing thereby maintainance efforts.

Systematic integration is achived by using the action dispatcher design pattern. The action dispatcher design pattern provides a single access point for communication with hypermedia services, selecting the appropriate action by dispatching centrally all incomming requests. Firgure 4 depincts the action dispatcher design pattern.

In Figure 4, all requests for hypermedia services are dispatched by the Dispatcher class that creates the appropriate operation that needs to be requested from structure server. Thus, every operation that is available by a particullar structure server is represented as a separate class. Each such class, in turn, extends a generic Action Handler class.

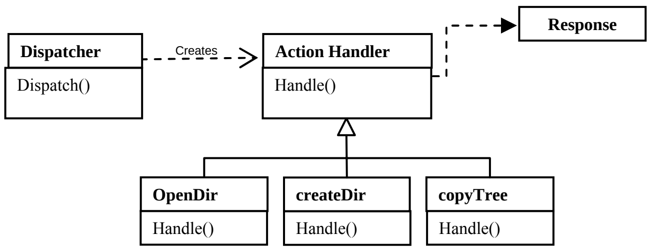

Figure 4: Dispatchign requests

Selection of the specific operation (or action) is done using a creational pattern (e.g. factory method).

3.2.2 Request chaining

At the web application layer and specifically during the handling of a particular user request, a number of hypermedia services need to be invoked sequentially – passing responses from one invocation to the other - to complete a user transaction. Moreover, situations arise where hypermedia and content services need to be invoked sequentially to produce the final response that will be sent back to the user. Similar invocations schemes are used within the web application layer (and not only in relation with Callimachus) such as validating user request using filters before invoking hypermedia services.

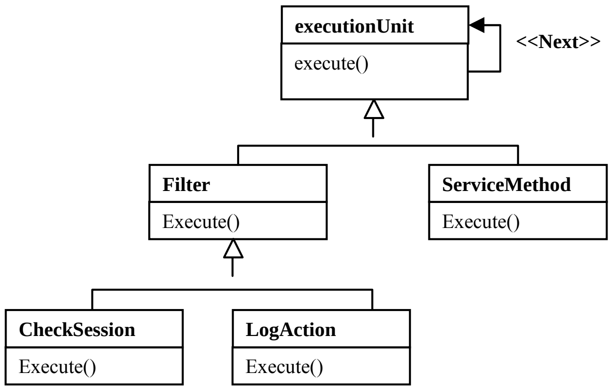

Figure 5: Sequential invokation of operations

Figure 5 depicts the design pattern to support sequential invocation schemes. Currently at the web application layer, two types of operations can be chained: filter and hypermedia service invocations. Actions that need to be invoked within such "chain" need to extend the appropriate class providing developers a convenient way to specify sequential invokation of services and operations in general.

4. CONCLUSIONS AND FUTURE WORK

In this paper we presented design patterns that address evolution concerns in web application that are based on SOA. In particular we described what design pattterns have been implemented within the Callimachus project – a CB-OHS- that provides hypermedia services to a broad range of clients including web applications. In Callimachus, design patterns are used to address evolution and maintenance concerns at the web application and hypermedia service layer. Although the design patterns mentioned are already known, we have discussed them in a service oriented context.

Future work includes identifying additional design patterns to address even more elaborate evlolution scenarios. We believe that design patterns have a particular role to play when building web applications on SOA.

5. REFERENCES

1. Ginige, A., Murugesan, S., Web Engineering: An Introduction, *IEEE MultiMedia*, 8(1), Jan.–Mar. 2001, pp. 14–18. *(CHI '00)* (The Hague, The Netherlands, April 1-6, 2000). ACM Press, New York, NY, 2000, 526-531.
2. Papazoglou, M. P., Georgakopoulos, D. (eds.), Service-Oriented Computing, *Communications of the ACM*, 46(10), 2003.
3. Agrawal, R., Bayardo, R. Jr., Gruhl, D., Papadimitriou, S., *Vinci: A Service-Oriented Architecture for Rapid Development of Web Applications*, in Proceedings of the 10th Int'l Conference on World Wide Web (WWW '01, Hong Kong, Hong Kong), 2001, pp. 355–365.
4. Wiil, U. K., Nürnberg, P. J., Hicks, D. L., Reich, S., *A Development Environment for Building Component-Based Open Hypermedia Systems*, in Proceedings of 11th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '00, San Antonio, Texas, USA), 2000, pp. 266–267.
5. Nürnberg, P. J., Leggett, J. J., Schneider, E. R., *As We Should Have Thought*, in Proceedings of the 8th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '97, Southampton, UK), 1997, pp. 96–101.
6. Tzagarakis, M., Avramidis, D., Kyriakopoulou, M., Schraefel, M., Vaitis, M., Christodoulakis, D., Structuring Primitives in the Callimachus Component-Based Open Hypermedia System, *Journal of Network and Computer Applications*, 26(1), January 2003, pp. 139–162.
7. Vaitis, M., Papadopoulos, A., Tzagarakis, M., Christodoulakis, D., *Towards Structure Specification for Open Hypermedia Systems*, in Proceedings of the 2nd Int'l Workshop on Structural Computing, Springer-Verlag LNCS 1903, 2000, pp. 160–169.
8. Wiil, U. K., *Multiple Open Services in a Structural Computing Environment*, in Proceedings of the 1st Int'l Workshop on Structural Computing (SC1, Darmstadt, Germany), Technical Report AUE-CS-99-04, Aalborg University Esbjerg, Computer Science Department, Denmark, 1999, pp. 34–39.
9. Beringer, D., Melloul, L., Wiederhold, G., *A Reuse and Composition Protocol for Services*, in Proceedings of Symposium on Software Reusability (SSR'99, Los Angeles, California, USA), 1999, pp. 54–61.
10. Buschmann, F., Meunir, R., Rohnert, H., Sommerland, P., Stal, M., *Pattern Oriented Software Architectures: A System of Patterns*, John Wiley & Sons, 1996.
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.
12. Hu, J., Schmidt, D. C., JAWS: A Framework for High-performance Web Servers, in Fayad, M., Johnson, R. (eds.), *Domain-Specific Application Frameworks: Frameworks Experience by Industry*, John Wiley & Sons, 1999.
13. Dart, S.: Configuration Management: the missing link in Web engineering. Artech House, 2000.
14. Karousos, N., Pandis, I., Reich, S., and Tzagarakis, M. (2003). Offering Open Hypermedia Services to the WWW: A Step-by-Step Approach for the Developers. In Proceedingss of Twelfth International World Wide Web Conference WWW2003, (Budapest, Hungary), pp. 482-489.
15. Wiil, U., Nurnberg, P., Evolving hypermedia middleware services: Lessons and observations. Proceedings of the Thirteenth ACM Symposium on Applied Computing (SAC 99), San Antonio,TX, US, Mar.,1999