Software architecture style for interoperable databases.

Radmila Juric[1]
Jasna Kuljis[2]
Ray Paul[2]

[1] School of Electronics and Computer Science, Westminster University
[2] Department of Information Systems and Computing, Brunel University

Software Architectural Style for Interoperable Databases

Radmila Juric[1], Jasna Kuljis[2], Ray Paul[2]
[1]*Cavendish School of Computer Science, Department of Information Systems, University of Westminster, 115 New Cavendish Street, London W1W 6UW*
juricr@wmin.ac.uk , priades@aol.com
[2]*Department of Information Systems and Computing, Brunel University Uxbridge, Middlesex UB8 3PH, UK*
Jasna.Kuljis@brunel.ac.uk Ray.Paul@brunel.ac.uk.

Abstract *We propose a layered and component based software architecture style which supports interoperability in multiple databases (DB). The architectural style's building blocks and its constraints are described and the deployment of two design patterns outlined. Components placed in our architectural layers exhibit a linear topology and request/reply processing style. The constraints include communications between components which are not in the adjacent architectural layers and extension of the intuitive many : one bindings between components towards many : many. We comment on similarities with mediation architectures and outline some implementation issues.*

Keywords. Software architecture style, DB interoperability, components.

1 Introduction

The problem of DB interoperability has been in the focus of research in the DB community since the early 1990s. There exist a numerous work which covers both theoretical and pragmatic issues that deal with semantics of the interoperability problem in multiple DB systems. This has been delivered through varying degrees and configuration of integration, interdependency and exchange amongst multiple DBs in order to achieve effective sharing and use of each other data and functionality [8]. The DB interoperability problem has been addressed through (a) migration between various DB systems (e.g. from relational and object DB) [13,21,27,31,32], (b) multidatabase and federated architectures as defined in [29] and exploited in works from [4,23,25,10,16] and (c) mediator paradigm from [34.35], which has culminated in research projects [28,22,1,6] and many more. Each of these approaches suffer from drawbacks, and today's trend in DB centric applications is (i) to allow the individual data structures to evolve naturally within their own environments, and (ii) to build/offer services that will provide transparent facilities across different DB systems [8]. A component based software architecture, which accommodates interoperation and allows extendibility and scalability of the multiple DB systems and data repositories, while preserving the autonomy of their individual elements, might be a solution to the DB interoperability problem.

In this paper we propose a software architectural *style* for interoperable DBs by exploiting the role of software architecture as defined in [30,3], and using a generic component-based software architecture model given in [20]. Our solution avoids integration, centralization of data and structures, and use of common database languages. The proposed architectural style resembles mediation architectures, but contains a distinctive set of constraints imposed on the nature of architectural components and the way they interact with other components within multiple DB system.

Section 2 details the reference architectural model, section 3 describes our architectural style and its constraints, section 4 comments on our architectural style and related work, section 5 outlines some implementation issues and section 6 concludes with the overview of future works.

2. A Layered Reference Model

The five layered reference architectural model for interoperable DBs from [20] is given in Fig. 1. We use a component-based technology [33] and proposed layering is based on how specific/general to our problem requirements each component is. We use layered architecture as described in [2] where layers are "allowed to use" public facilities of the nearest lower level.

Consequently, the usage of layers in layered architectures flows downwards.

The application layer is the most specific layer, whose components are responsible for:

- providing GUI functionality
- managing interaction between users and software layers,
- analyzing the functionality of user's requests imposed on multiple DBs and routing such requests towards appropriate components of the lower layer(s),
- managing value added services, such as querying metadata or adding user's intervention, which might be essential when resolving heterogeneity problems in multiple DB systems (particularly important when dealing with semantic and schematic heterogeneity, various expressiveness of database languages and smilar)

Components from this layer encapsulate user/application specific code, which may be distinct and not re-usable in, or interoperable with, any other applications. For example: requests for DB retrieval written in a specific DB language might not be reusable across all applications that use a multiple DB system.

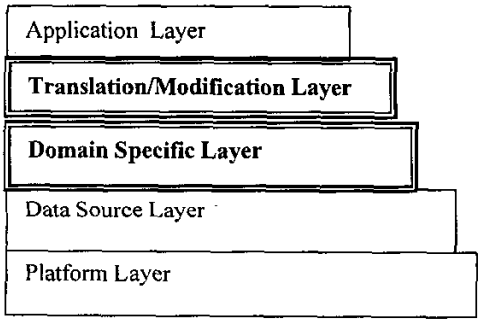

Figure 1. Layered software architecture ref. model

The translation/modification layer is a core layer which contains a family of components that are responsible for translating the user's request to a targeted operating environment. For example: we may need to translate a relational SQL-query into a set of different joint queries that range from object to XML data retrieval; or we can translate an existing relational schema declaration into class declarations of an object DB, etc.

Components from this layer encapsulate a code that can be shared amongst requests originated in various DB applications, i.e. the code can be used by a family of related applications. For example all applications that place relational SQL-like requests may use the same components from the translation layer in order to translate the request to a set of different queries to match the targeted DB environment.

Important: the code in these components is dependent on translation needs originated in the application specific layer. Hence services from the components in this layer might not be required in some systems, such as a centralised DB system, or fully replicated distributed DB with shared DB architecture, or multiple DB systems deployed with the same DB technology and which exhibits no semantic and schematic heterogeneity.

The domain specific layer is a core layer which manages user's workspace and contains a family of components that are responsible for:

- adhering to a particular application domain in order to manage users' requests
- implementing the functionality of user' requests by applying it to domain specific components that are derived from general-purpose persistent components from the data source level below.

Components from this layer encapsulate a code that may be used from different places within the same application and by a family of related applications. For example: components housed within the domain specific layer may implement functionality of (a) joining a relational table and XML document for retrieval or (b) executing a family of data definition statements in order to create various data structures or schemas simultaneously across multiple DBs, etc.

The data source layer handles entity business logic ('entity' taken from the EJB technology [36]). It is made up of components that provide persistence and programming infrastructure services for general-purpose persistent components. Components from this layer encapsulate potentially reusable code across many application domains.

The platform layer accommodates components that underpin the application and which include everything from operating systems, Database Management Systems, GUI class libraries and similar.

3. Our Architectural Style

A number of distinct architectural styles have been given in [30]. They all describe their building blocks through: (i) the nature of architectural components i.e. the nature of their

computation, (ii) the way they interact with other components when composing a system and (iii) constraints on the way this composition is done. Many real life problem domains combine more than one architectural style into a specific one, by incorporating useful aspects from several of them.

3.1. Building Blocks of our Architectural Style

The building blocks of our architectural style are shown in Fig. 2. We identify some primitive and composite components through their participation in our architectural layers, we give their substructure if they are composite and declare what each component implements.

Component ***A***, *Analyzing User's Request* is a primitive component, but it implements extensive functionality. It analyses user's requests and determines in the following order:

a. *translation requirements*: e.g. the user's request may be executed within its own operating environment and there will be no need for translation services or v.v.
b. *functionality*: e.g. the request could be for the creation of a new structure or DB element, or manipulation of existing DB elements etc.

Translation of a particular user's request is required from components Tr_i in the translation layer only if the user's request is to be executed outside the environment where it originates.

- If the received request needs translation, we determine its functionality, before routing it towards a particular Tr_i component.
- If the received request needs NO translation, its functionality has to be determined first in order to route un-translated requests directly towards an appropriate set of D_1', ...D_n' components in the domain specific layer that implement this functionality.

Components Tr_i belong to a family of primitive components where each of them implements a different algorithm for translation of users' requests to a targeted DB environment. In other words, each Tr_i component provides different implementations of the same behaviour (i.e. translation), where the received request and user's understanding of the problem decide the most suitable implementation. Note: translation algorithms might include value added services such as user's interventions for resolving DB semantic and schematic inconsistencies during translation (if this has not been done by the *A* component).

Components D_1' ...D_n' belong to a family of D_i' components where each D_i' component may encapsulate any combination of D_i components. Each primitive D_i component is a subset of general-purpose persistent component from the data source layer. Each D_i' implements certain functionality, which is required by the *application specific* layer, and which can be performed on any combination (D_1 , D_2, ... D_n) of persistent data components. We may group the functionality of received request into various categories. One grouping is suggested in b. above: *data definition* (DD), *data manipulation* (DM) and *data entry* (DE). Note that the *functionality* of each received request is determined regardless of its need for translation.

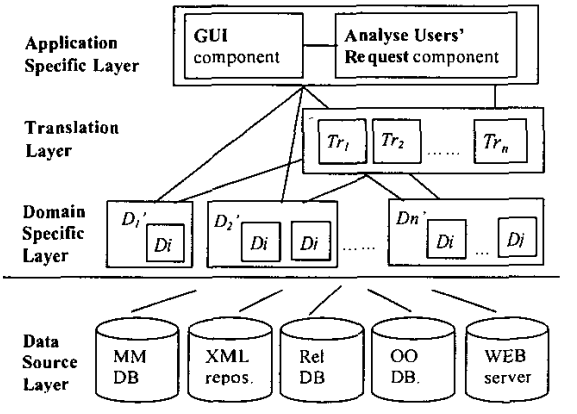

Figure2. The building blocks of our architectural style

3.2. Constraints on our Architectural Style

The layering principle given in the reference model from Fig. 1 aims to achieve a certain level of flexibility, reusability and extensibility of the architectural solution for the interoperable DBs. This principle of software layering is not new. It has been used as a reference model in network layering [17] and it is also a recognized software architectural style [7].

3.2.1. Our Architectural Style and the ISO model

In the ISO reference model [17] software modules from one layer communicate directly only with the layers above and below it. We can partially map our reference architecture and the ISO model: the presentation and session layers from the ISO model may accommodate our core

layers of *translation* and *domain specific* components. However, there is a difference:

Our architectural components communicate from one layer to any other lower layer, hence communication is not limited to adjacent layers.

The ISO reference model assumes that heterogeneity is managed at the lower levels of the model and that the higher levels need not be aware of any lower level differences. However, our architecture produces an explicit architectural support for managing heterogeneity through the extensible set of components from the *translation* and *domain specific* layers positioned immediately below the top level. Furthermore, the level and nature of heterogeneity directly determines the communication pathway between components (i.e. layers), which is built-in into our architectural model.

3.2.2. Combining Architectural Styles and Connecting Layers

In spite of organising our components in layered topology, Fig. 2 shows that we allow a certain level of 'a one way data-flow through a network of filters', i.e. linear topology within the application specific layer, e.g. the user's requests are 'filtered' through the 'analyse users' request' component before we proceed to any other layer. However, the connection between layers might exhibit the request/reply processing style. Our architectural style from Fig. 2 allows components to require and offer services from components that are not in adjacent layers and the execution order is partially determined by a format of incoming users' requests (see 3.2.1). This points towards the star topology, where requests and replies are bound according to components needs.

3.3. Using Design Patterns

The *Strategy* pattern is used within the core layers when generating a family of translation algorithms encapsulated in the Tr_i components and a family of D'_i components that implement the functionality of user's requests. In both cases we provide different implementations of the same behaviour where the user's request (and user's understanding of the problem[1]) decide the most suitable implementation. The *Chain of responsibility* pattern is used throughout the model: all tasks are handled by the different components in the chain (A, Tr_i, D'_i) and forwarded ultimately to the component, which executes it. We allocate components dynamically in the chain, i.e. according to the functionality of user's requests, and bypass some (e.g. Tr_i components).

Both patterns help us to vary one part of our architectural structure independently to some other parts, making our system more robust to change, addressing reusability and achieving extensibility. We argue that:

1) translation algorithms, as parts of software that are likely to change in future (e.g. to be extended, optimized or to be changed completely) are isolated;
2) we define as many different variants of the same algorithm as possible, i.e. a family of related algorithms;
3) we generate new algorithms through previous knowledge/experiences and properties of operating environments as in evolutionary programming, or machine learning techniques;
4) the user chooses the most suitable algorithm, i.e. different tactics according to trade-offs when executing DB transactions (N.B. the user is aware of different algorithms – the requirement of the *Strategy* pattern);
5) loose coupling between components allows easier modification/extension of the system's functionality:
   i. some components can be bypassed if we do not need them and
   ii. the implementation code of components in the *Chain of responsibility* pattern might be simplified, because we do not need to know which component in the chain is going to handle a particular request.

3.4. Summarizing the Characteristics of our Architectural Style

1. We separate components into layers according to their specificity within the application. Layer ordering is based on compile time dependencies. The higher a component in the model is, the more specific it is and the more dependent on other components (i.e. less reusable) and v.v.
2. Our core (*translation and domain*) layers push away application specific requirements from generic functionality of data sources

[1] This user intervention is very important in building an automated tool for translation algorithms' support.

and computing platforms, making systems more adaptable to changes.

3. The content of a particular component may be decided upon which layer it is appropriate to reside, i.e. knowing the layer in which the component resides, we know which services it offers.
4. There is a possibility of *extending families of core layer components* without affecting existing components in the same and adjacent layers. In addition, we may *generate in advance core layer components* to suit new requirements/applications.
5. Our layering architecture allows components from a particular layer to use services of components from *any other layer* and not only from adjacent layers. Components within a particular layer can also use each other's services.
6. When binding services we allow that an intuitive *many : one* relationship between requirements and provisions is extended towards *many : many*, i.e. each required service may be bound to more than one provided service.
7. Composite components of our core layers may contain a variable number of primitive components and consequently a variable number of interfaces. They are determined by
   i. the functionality that a particular family of components implements (see (b) in 3.1);
   ii. the desired level of granularity of individual primitive components.

   Aiming to generate fine-grained components with discrete functionality and low overhead will increase the number of components needed within the core layers and v.v.

4. Comments on our Architectural Style and Related Works

Our generic reference architecture from Fig. 1 and the architectural style from Fig. 2, resemble mediation architectures introduced in [34,35,36] elaborated in [37,38], and compared in [5]. Mediator software modules, placed in between data resources and applications, provide *intermediate services* in heterogeneous, autonomous, distributed and evolving information systems. Their aim is also to *'avoid the integration of data resources'* and to give an *'integrated and consistent view'* of heterogeneous and distributed data/information available, hence avoiding any centralization. Our `Analyze user's request` component may play a role of a '**mediator**' in order to chose the best pathway through the layers when implementing required functionality. However, the mediation [37] assumes that the retrieved data from heterogeneous environments is to be abstracted and transformed in order to be *integrated* through matching keys, and processed to *increase the density of information.* Furthermore, the mediation architecture, which extends the client-server model includes *integrators*, whose role is to *combine resources* that could be shared and generalized. Wrapping of the existing data repositories or DB applications is an essential interface/input towards a mediation process: wrappers deal with data model and platform heterogeneity and mediators' role is to resolve semantic and schematic heterogeneity [5].

Many research projects based on mediation have been developed: Garlic [28], Information Manifold [22], SIMS [1], InfoMaster [9], DIOM [24], COIN [15] etc. Our architecture overlaps with the TSIMMIS [6,14]:

- information from heterogeneous and autonomous information sources are combined without having a global view of integrated information or a global database schema;
- translators are used to convert data models and queries;
- mediators can automate information integration and use expert knowledge and value added services in order to process specific information.

However, they differ in some aspects:

- translation of information, i.e. data model and queries, into a common object model called OEM is essential in TSIMMIS;
- queries written in OEM are submitted to a mediator(s) and successively translated into local queries (i.e. queries of local database sources) and the user is expected to write queries in OEM;
- browsing of information and/or information exchange is allowed (OEM is an information exchange model), but no update or creation of new data structures are considered.

5. Implementation Issues

We are confident that our architectural style can be evaluated. Our layered architectural model of extensible set of domain specific components has been applied in the problem of

interoperability of medicinal product evaluation practices in healthcare systems [19]. We are currently implementing a simple example of DB interoperability as an EJB [11] application: *a user's request for creating a new DB structure, written in a relational SQL, results in simultaneous creation of three different structures: relational table, a class for the object database and a DTD for an XML document* [20]. It is essential that the user is not required to use any other DB language except a relational SQL, and he/she might not be aware that any translation between relational SQL creation command and class/DTD declaration statements is taking place. This trivial example is needed at our postgraduate tutorials within the IS curriculum, where students are prepared to discuss the DB interoperability problem, to evaluate component technologies and to design and deploy an example of distributed application within the J2EE platform available at the University. Students have designed a multi-tier EJB application, with the Web container which hosts Web components responsible for handling the GUI, and the EJB containers, which host the application components: A, Tr_i and D_i':

(a) The Tr_i components are model examples of stateless session beans: each Tr_i contains a simple request and response functionality (e.g. translation from relational to the OO DD statement). They call only one method per session, operate on arguments that client passes to it (e.g. table name, attribute names and types for Tr_i), they can be used sequentially by many different clients and need no tailoring to suit a specific client. Students have found the `converterEJB` example from the J2EE Reference Implementation at [12] very useful when coding the Tr_i methods of the bean class and its remote interfaces.

(b) The D_i' components have been designed as statefull session beans, because they contain more complex interactions and maintain conversational state between a client and EJBs, they may call more than one method per a session and consequently may manipulate one or more entity beans within a single session. These session beans may access DBs using JDBC and the J2EE connector architecture, which may eliminate any need for entity beans.

We outline two 'controversial' issues:

(1) The PointBase Server 4.2 [26] installed within the Sun ONE Studio 4 IDE for implementing entity beans triggered different views on (a) the extent to which the EJB platform addresses the DB interoperability problem itself and how it might picture our architectural style as 'redundant' and (b) the J2EE Reference Implementation's assumption that legacy systems are always from relational environments [12].

(2) The A component's functionality, designed as a statefull session bean provoked discussions on:
   a. the role of value added services, ontology, metadata and user's intervention when addressing heterogeneity in multiple DB systems;
   b. the decision of giving a control of the whole application to the A component which manages interaction between all layers of the application from Fig. 2 and equalizing it with the role of controller objects [18] rendered from the requirement analysis model of such an application.

All of these will be addressed in future works.

6. Conclusions

This paper addresses the DB interoperability problem through component based software architectural style and moves away from migrations between technologies, federated architectures, multi-database languages and similar works from the 1990s. Our solution uses a layered software architecture, populated with components that exhibit linear topology and request/reply processing styles. The most important constraints allow components to communicate with any layer, i.e. their communication is not restricted to adjacent layers only, and their binding is extended towards *many : many*. In spite of these two distinctive constraints, our software architectural style resembles mediation architectures and their ideas of how to tackle the DB interoperability problem. Various mediators, adapters, integrators and wrappers found in mediation architectures, all serve to ease the interoperability of today's DB applications, by addressing at various levels their heterogeneity, autonomy, need for evolution and unavoidable distribution. Some of our components can be mapped to and play a role of mediators and adapters/wrappers. From this respect our architectural style does not represent a significant change in the way the DB interoperability is

tackled today. However, our contribution is centred on the following two:

1. We address the problem of semantic and schematic heterogeneity, needs for value added services, involvements of metadata and access to possible ontology at the top most layer of our architecture, i.e. within component A. In other words, our mediation or modifications or wrapping, happen very early, i.e. users will have to deal immediately with certain kinds or levels of heterogeneity, which we see as very reasonable. If we allow a multitude of DB languages, schemas or DB system, to be available for today's DB applications, we either pay a conceptual price for this or seek users interventions when trying to automate or solve problems arising from heterogeneities.
2. We have proposed in section 3 the analysis of functionality of user requests in order to determine the best possible pathway for our components' executions. The reasons are in
   (a) categorising the functionality of user request may affects the granularity of our components and simplify the implementations of algorithms of components from the translation and domain specific layers [21,20].
   (b) contextualising and contracting software components as in [33] may also have an impact on the granularity of our components. This will also allow us to go beyond published required/provided services of our components. We may specify conditions attached to components' contracts that extend the management of structured and semi-structured data towards continuous data streams of multimedia applications.

In our current work we map the functionality of user requests issued upon multiple DBs to components' contracts. We contextualize components within such a contract through categorizations of user requests.

References

[1] Arens Y, Cluet S, Milo T. Query Processing in the SIMS Information Mediator. In Tate A, editor. Advanced Planning Technology, AAAI Press, 1996, 61-69.

[2] Bass L, Kazman R. Architecture-Based Development, Technical Report, CMU/sei-99-TR-007, esc-TR-99-007, 1999.

[3] Bass L, Clements P. Kazman R. Software Architecture in Practice, Addison Wesley Longman Inc., 1998.

[4] Bukhres O, Elmagarmid AK, Editors. Object-Oriented Multidatabase Systems: A Solution for Advanced Applications, Englewood Cliffs: Prentice Hall, 1996.

[5] Busse S, Kutsche RD, Leser U, Weber H. Federated Information Systems: Concept, Terminology and Architectures, Technical report No 99-9, TU Berlin, Germany,1999.

[6] Chawathe S, Garcia-Molina H, Hammer J, Ireland K, Papakonstantinou Y, Ulamn J, Widom J. The TSIMMIS project: Integration of Heterogeneous Information Sources, Proceedings. of the IPSJ Conference, Tokyo, Japan, 1994, 7-18.

[7] Clements, P, Bachman F, Bass L, Garland D, Ivers J, Little R, Nord R, Stafford J. Documenting Software Architecture: Views and Beyond, Addison Wesley 2003.

[8] Dulay N, Juric R. On Interoperability in DB Environments, An Analysis of Past and Current Trends in the DB field, under review for Journal of Integrated Design and Process Science.

[9] Duschka OM, Genesereth MR. Querry Planning in InfoMaster, In Proceedings of the 12th ACM Symposium on Applied Computing, 1997, San Jose, CA.

[10]Elmagarmid M, Rusinkiewicz A, Sheth A, editors. Management of Heterogeneous and Autonomous Database Systems, Morgan Kaufman Publishers Inc., San Francisco, 1999.

[11]Enterprise JavaBeans Specification http://java.sun.com/products/ejb/docs10.html

[12]EJB Reference Implementation available at http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/GettingStarted.htm http://www.pointbase.com/support/releasenotes.html],

[13] Fong J. Converting Relational to Object-Oriented Databases, *ACM SIGMOD Record 1997*, 26(1), 53-58.

[14] Garcia-Molina H, Hammer J, Ireland K, Papakonstantinou Y, Ulamn JD, Widom J. Integrating and Accessing Heterogeneous Information Sources in TSIMMIS, In Proceedings of the AAAI Symposium on Information Gathering, Stanford CA, 1995, 61-64.

[15]Goh CH, Madnick ME, Siegel MD. Context Interchange: Overcoming the Challenges of Large Scale Interoperable Database Systems

in Dynamic Environments, In: N. Adam, B. Bhargava, Y. Yesha editors. Proceedings of the 3rd International Conference on Information and Knowledge Management, ACM Press, 1994, 337-346.

[16] Hull R. Managing Semantic Heterogeneity in Databases: A Theoretical Perspective, ACM/PODS 1999, or http://www-db.research.bell-labs.com/user/hull/pods97-tutorial.html .

[17] ISO standards available at http://www.iso.ch/iso/en/CatalogueListPage.CatalogueList .

[18] Jacobson I, Booch G, Rumbaugh J. The Unified Software Development Process. Reading, MA: Addison Wesley; ACM Press, 1999.

[19] Juric R, Juric J. Applying Component Based Modeling in the Process of Evaluation of Medicinal Products, in Proceedings of the 5th International Conference on Integrated Design and Process Technology, 2002, Pasadena, CA.

[20] Juric R, Kuljis J, Paul R. A Software Architecture to Support Interoperability in Multiple Database Systems. In Proceedings of the 22nd IASTED International Conference on Software Engineering, February 2004, Innsbruck, Austria.

[21] Juric R, Martin N. DBRE and Migration from Relational to OO Databases, In: Kalpic D, Hljuz-Dobric V, editors. Proceedings of the 20th International Conference on Information Technology Interfaces;1998 Jun, Pula Croatia. Zagreb: SRCE University Computing Centre, University of Zagreb; 1998, p. 323-334.

[22] Kirk T, Levy AY, Sagiv Y, Srivastava D. The information Manifold. Proceedings of the AAAI '95 spring symposium on Information gathering from Heterogeneous Distributed Environment, p.85-91.

[23] Litwin L, Mark L, Roussopoulos N. Interoperability of Multiple Autonomous Databases, ACM Computing Surveys, 1990, 22(3), 267-293.

[24] Liu L. Pu C. Query Processing in DIOM, IEEE Quarterly Bulletin on Data Engineering, Special Issue on Improving Query Responsiveness, 20(3) 1997.

[25] Oszu TM, Valrudrez P. Principles of Distributed Database Systems, Prentice Hall Inc., 1999.

[26] PointBase Developers Guide, available at http://pointbase.com/support/releasenotes.html

[27] Ramanathan S, Hodges J. Extraction of OO Structures from Existing Relational Databases, SIGMOD Record 1997, 26(1), 59-64.

[28] Roth MT, Schwarz P. Don't Scrap it, Wrap it! A Wrapper Architecture for Data Sources, Proceedings of the 23rd VLDB Conference, Athens, Greece, 1997, 226-275.

[29] Shetsh A.P, Larson JS. Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases, ACM Computing Survey, 1990, 22(3), 183-236.

[30] Shaw M, Garland D. Software Architecture, Perspectives on an Emerging Discipline, Prentice Hall, Inc., 1996.

[31] Soudi A, Nachouki G, Briand H. Relational Database Reverse Engineering: A Knowledge-Based Approach, In: Proceedings of the 3rd International Conference on Object Oriented Information Systems, London, UK, 1996, 180-194.

[32] Soutou C. Relational Database Reverse Engineering: Extraction of an IFO2 Schema, in 1995.

[33] Szypersky C. Component Software–Beyond Object-Oriented Programming, Addison Wesley, 1998.

[34] Wiederhold G. Mediators in the Architecture of the Future Information Systems, IEEE Computer, 25(3), 1992, 38-48.

[35] Wiederhold G. Intelligent Integration of Information, SIGMOD Record, 22(2), 1993, 434-437.

[36] Wiederhold G (1994) Interoperation, Mediation and Ontologies, *Workshop on Heterogeneous Co-operative Knowledge-Bases,* ICOT, Tokyo, Vol. W3, 33-48.

[37] Wiederhold G, Genesreth M. The Conceptual basis for Mediation services, IEEE Expert, 1997.

[38] Wiederhold G. Mediation to deal with Heterogeneous Data Sources, 1999 http://www-db.stanford.edu/pub/gio/1999/Interopdocfigs.html