# MODELS FOR INTERACTION, INTEGRATION AND EVOLUTION OF PRE-EXISTENT SYSTEMS AT ARCHITECTURAL LEVEL

## *Designing Systems for Changing Conditions*

Juan Muñoz López, Jaime Muñoz Arteaga, Francisco Javier Álvarez Ramírez, Manuel Mora Tavarez
*Universidad Autónoma de Aguascalientes, Av. Universidad 1309, Fracc. Campestre, 20190, Aguascalientes, Mexico*

Ma. Lourdes Y. Margain Fernández
*Universidad Politécnica de Aguascalientes, Av. prol. Mahatma Gandi Km. 2, Sn. Fco. del Arenal, Aguascalientes, Mexico*

Keywords: Software architectures, architectural models, architectural patterns, integration, interaction, evolution.

Abstract: This paper describes a set of models that may serve as the basis for the creation of architectural patterns for interaction, evolution and integration of pre-existent systems. Proposed models are based on an identification of system's specialized components for operation, control and direction making easier to find connecting points between systems. This set of models covers some rationality needed to adapt pre-existent systems to an evolving environment where organization and technologies are under continuous change.

## 1 INTRODUCTION

New systems must be integrated into heterogeneous computational environments in continuous change and coexist with pre-existent systems. Under this precept, architectural design must incorporate capacities of interaction, integration and evolution into the systems; this has been a preoccupation of many works in software engineering area.

Interaction refers to information exchange between systems. An effective interaction needs of certain basic conditions: 1, all participating systems must understand information in the same way; 2, one or more systems must answer with adequate information to requests from other systems; and 3, information must be recovered without distortion.

Integration occurs when some degree of control yielding from one or more systems is added to the interaction operation. An integrator system will get some kind of control over the integrated systems modifying their execution flow.

Evolution of a system is produced by applying maintenance to its components modifying or replacing them. Evolution is used to increase or decrease functionality adapting the system to new environmental and organizational requirements.

An architectural structure may promote or impede system's capacities of interaction, integration an evolution. This paper proposes a set of models that will provide some criteria to ensure the incorporation of such qualities.

Rest of the paper has been organized as follows: Section 2 shows the challenges that must be solved to get interaction, integration and evolution in a defective architectural design. In section 3 we make a review of related works which deals with these problems. Section 4 shows a proposal of a set of models oriented to help to include the mentioned qualities in architectural models. In section 5 we present a system which applies the models. Finally, section 6 contains some conclusions and recommendations for future work in this mater.

## 2 CHALLENGES

Almost always, design of systems that will be incorporated into a computing environment should consider their interaction with existing systems. Many of them are legacy systems: "large systems delivering significant business value today from a


Muñoz López J., Muñoz Arteaga J., Javier Álvarez Ramírez F., Mora Tavarez M. and Lourdes Y. Margain Fernández M. (2008).
MODELS FOR INTERACTION, INTEGRATION AND EVOLUTION OF PRE-EXISTENT SYSTEMS AT ARCHITECTURAL LEVEL - Designing Systems for Changing Conditions.
In *Proceedings of the Third International Conference on Software and Data Technologies - SE/GSDCA/MUSE*, pages 220-227
DOI: 10.5220/0001891702200227

substantial pre-investment in hardware and software that may be many years old" (O'Callaghan, 2000).

Sooner or later, legacy systems must be taken off because their architectural constitution will make their adaptation very costly and difficult. However, since those systems have an important role for the organization, we need to provide a framework to let them work while we gradually replace them.

An architect must have a wide vision to model systems that will grow up along with user's requirements. System qualities are part of stakeholders' requirements and they are in a continuous evolution too. A lot of systems lack of a good design because they has been built to accomplish only a set of initial requirements.

As time passes by, users will require more functionalities and communication capacities from the system. Also, as the computational environment grows, it becomes more complex, so we need to increase governability and usability in each system.

When we add new capacities to a system, we provoke an evolution on it. The system is adapted to new circumstances and requirements by applying maintenance. Maintenance adds complexity to the system, so later adaptations will be harder and more expensive to do.

Software reengineering is used to reduce complexity and to increase evolution capacity of a system by refactoring software code; but, it doesn't help to change its architecture or functionality (Sommerville, 2004). This strategy is associated with some kind of preventive or perfective maintenance; it's a costly and difficult evolution strategy (Pressman, 2004).

Re-engineering of legacy systems will not always be possible because of different factors such as absence of source code, lack of tools to recompile it, lack of skills and knowledge to do it, etc.

A similar situation applies to interoperability. Many systems have been designed to work separately, so their communication capacity is very limited. Systems that aren't prepared to interoperate must incorporate complex and inefficient mechanisms to exchange information.

We add interaction capacities to exchange information and functionality between independent systems in a more effective way.

Integration adds an additional coordination element to interaction. In this case a system takes partial or total control over functionality and information of other systems. System integration can bring us various benefits such as reduction of software complexity, increment of systems governance and establishment of a framework for evolution of the computational environment.

System integration is a way to simplify a computing environment. This strategy helps to improve software qualities like: efficiency, governability, maintainability and usability.

To achieve an extensive capacity of interaction, integration and evolution of systems using any available technology requires models to support the design of software architectures.

If we don´t have a good architectural design since the beginning, we will need to change the core structure of the system to add or expand functional and non-functional characteristics in critical systems that cannot be replaced in a short time. This is a difficult task because software architects will find complex structures, different programming styles, ignorance and lack of documentation of how the whole system works, among other problems.

With architectural models we can share conceptual rules to develop new patterns that will help to make easier the task of designing and redesigning software systems.

Creation of reference architectural models to facilitate designing efforts will require establishing a practical method for identifying generic architectural elements in existing systems to develop decision rules and to decide which will be the best suited model under certain conditions.

# 3 BACKGROUND AND RELATED WORK

Evolution is required to maintain systems operating in a changing environment. Robertson applies Dynamic Object Oriented Programming, specific domain programming languages and reflection to change system's behaviour at runtime (Robertson, 1997). He also says that connecting legacy systems helps to replace them by doing parallel deployments.

In the work of Ziegler and Dittrich (Ziegler & Dittrich, 2004) we can find a set of approaches that have been used to integrate information from different systems; such as: common user interfaces, applications, middleware, global views of data, common data storage, and so on. This work makes special emphasis on the need for adequacy of semantics, which certainly is a problem to be solved because we need to establish a comprehensive way to integrate components into an architectural model that will carry out contextualization tasks, a topic that is not addressed in that document.

Architectural integration of large systems has been studied by Garland and Antony (Garland & Antony, 2003). They recommend two strategies, one

based on data integration and the other based on executable components integration. However, they don't describe an architectural model to help us to figure out how such integration must be done.

Cornella (Cornella-Dorda et al., 2001) has proposed a method to modernize legacy systems code. It's based in a graphical diagram which puts in a tree every element of a program. Purpose of the map is to classify software parts as roots, leaves, nodes or isolated elements and make easier to develop a plan to carry out their migration.

Spaghetti code of legacy systems may make us fall into a tough identification job. It will be not easy to determine is a portion of code makes calls or if it's called, but often occurs that both things happens. Cornella's technique focuses on code migration, but not in architectural evolution.

The three tier architecture pattern has been extensively used for building a lot of systems (Trowbridge et al., 2004). This style identifies presentation, business logic and data as specialized components. Separation of data and presentation from business logic is convenient when we want to integrate different commercial components like browsers or database management systems (DBMS) that can be exchanged in a quite easy way.

However, using this approach doesn't give us sufficient elements to explain how to reuse functionality and data from different applications. We can create a common interface or use a single SMBD, but we need to explain who they will structure and contextualize information that must be understood by participating systems.

The work of Keshav et al. (Keshav et al., 1999) depicts some interesting elements needed for the architectural interaction of systems. He classifies them as: translator, controller and extender. The first element converts data and functionality between systems without changing their context; the second, coordinates movements of information under a predefined process; and the third one adds new functionality and features.

Elements described by Keshav et al. are necessary for interaction and integration; however, their work doesn't show how these components must be combined in a system. The work also shows how to make information exchange but it doesn't deals with systems integration.

The pattern established by the Model View Controller (MVC) makes a classification of interactive applications based on three areas: process, entries and exits (Buschman, 1996). For its implementation, the system is divided into three components: A model that contains functionality, data views that display information to the user and drivers that handle input from the user.

The MVC pattern is widely used to explain behaviour of interactive applications based on events reaction, but non interactive legacy applications cannot easily be defined under this logic.

A methodology that addresses integration of legacy systems is MEDARISH (Muñoz et al., 2006). This methodology describes a method that goes from a validated and structured set of requirements and constraints to a reference architecture model. Also, it covers the design of software architectures based in previous reference architectures, reference models and patterns, and makes emphasis in the integration of pre-existent systems.

# 4 A SET OF ARCHITECTURAL MODELS

Our proposal describes analysis criteria elements and a set of models to help rationalization tasks for interaction, integration and evolution of pre-existent systems when designing new architectural models. These rationalization tools have been conceived to be incorporated into MEDARISH methodology to be used in the architectural design process. These conceptual tools used in combination with other architectural artefacts reinforce system adaption capacities to meet future stakeholder requirements and environmental conditions.

## 4.1 System and Components Characterization

We can find three basic functions in a system: control, direction and operation. Control is needed to coordinate all elements of a system. Direction sets behavioural rules and constraints in the system as response to internal and external conditions. Operation is the group of elements which develop and deliver system products. These functions will be called "specializations" in this paper.

We can represent these specializations with UML stereotypes (see Figure 1) to represent basic components in the design of software architectures.

A component can represent a broker, a web service, a COTS application, a wrapping component, a class, a set of libraries or any other artefact. Components represented in this paper are independent of any existent technology or standard.

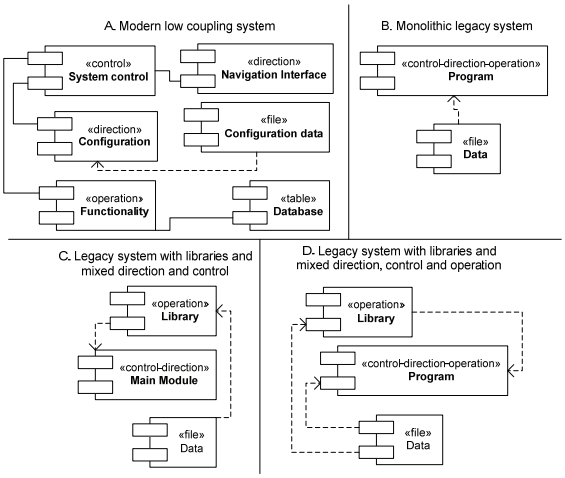


Figure 1: Generic System types.

Control components are in charge of managing logical conduction of a system, per example menus calling each system option. Direction components provide operational parameters and context information to control components; configuration modules, dialog boxes and menus are examples of them. Operation components collect data, make processes and deliver systems products, examples are: input screens, processing methods, output screens and reports, among others.

Permanent and semi-permanent data stored in files or database structures are taken in a separate way; they are used by functional components to develop its operations. For abstraction purposes, a database management system (DBMS) is not explicit showed in the diagrams used in this paper because it's considered only as a mean to access information. Stored procedures and functions would be considered part of operation components.

It is worth mentioning that each component could be composed by more than one architectural element in a real design.

Proposed taxonomy can be used in a recursive way. At a program level, an OOP application has a main method which exercises control when calling other methods; constructors and attributes can be perceived as direction elements; and finally, rest of methods are for operation. The model can also be applied to programming language elements at a lower abstraction level or to functions of systems that integrates big computational environments when working at a high rationalization level.

In figure 1A we have represented a generic system with separated and modular specialization of its main components, its structure makes easier to find connection points, necessary to implement interaction and integration.

System's logic for decisions resides in the "System Control"; it will make calls to "Functionality Components" to develop tasks and deliver products. Behaviour and decisions are based in pre-programmed rules from the "Configuration Component" and from the environmental feedback obtained from its "Navigation Interface".

Initialization data is stored in "Configuration Files". Some systems allow modifications of these parameters at runtime in answer to user feedback or changing environment variables as perceived by system's interfaces.

A monolithic system (Figure 1B) is composed only by an executable with mixed specializations and data files; it doesn't have modular components. This is a common architecture of legacy systems.

Some big sized legacy systems have libraries to implement part of its functionality (Figures 1C and 1D); this technique helps to put operation functions into a separate component. In the first case, the system has a main module with mixed direction and control and a separate module with all operational functionality. The second one represents a more common practice; part of the operation resides in the libraries and the rest in the main program.

Libraries have been commonly used since structured programming epochs to develop functionalities managing resources like memory, or making some kind of reuse in different programs.

## 4.2 Architectural Models

The set of models explained in the document address some cases of systems evolution, interaction and integration. For practical reasons we have classified them as:

1. Interaction with legacy monolithic systems using data transformation components.
2. Integration of legacy systems that have libraries.
3. Evolution of legacy systems using replacement libraries.
4. Transparent modular systems interaction.
5. Evolution by shared use of components with enhanced functionality.
6. Integration of systems by direct calls to functions in operation components.
7. Integration by coordination of pre-existent systems.
8. Full integration and evolution of pre-existent systems.

Proposed taxonomy is based in some common cases where evolution, interaction and integration is needed; but, this is a first approach and in a future

the set might be expanded with new cases that probably are not listed here.

### 4.2.1 Interaction with Legacy Monolithic Systems using Data Transformation Components

When dealing with a monolithic system, if we don't have access to source code or tools used for its compilation we will not have practical means to modify it. We only can interact with the system through its information files.

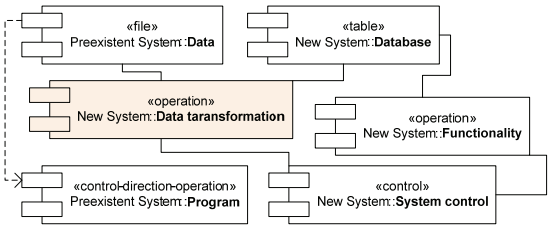


Figure 2: Interaction between a new and a monolithic system.

To obtain interaction it's necessary to create a bridge component to communicate information from the legacy system to the new one. This architectural element can be generically called "Data Transformation Component" (see Figure 2).

Data transformation component converts data from the legacy system file structure to the data structure of the new system. Perhaps this work will go far from adjusting structures because we must transform and standardize information semantics to adapt data from the context of one system to the other in order to make it useful.

This component could also have additional capacities like information transport between files and databases from both systems; and information enrichment, mixing and contextualization of data to support new functionalities. Also, when sending information to the legacy system it would be necessary to make arrangements to respect legacy validation rules to avoid future errors at runtime. This could be not feasible if we don't know validation rules for data of the legacy system.

This interaction mechanism gives us some advantages, like: creation of functionalities in the new system to substitute those similar in the legacy one; expansion of capacities in the new system to answer to users' and organization's needs; and establishment of a practical path for gradual replacement of legacy systems, by implementing "parallel" functionalities.

### 4.2.2 Integration of Legacy Systems that Have Libraries

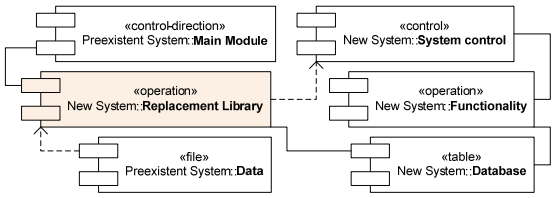


Figure 3: Integration of a legacy system using replacement libraries.

We can make a replacement library to include new functionalities in a legacy system. If we share control of this library between both systems (see Figure 3); this will result as an integration mechanism because the legacy system is yielding control of its functionality to the new system.

Replacement libraries can make unnecessary to develop a transformation component, because their operations can be integrated in the library.

### 4.2.3 Evolution of Legacy Systems using Replacement Libraries

If operative specialization of a legacy system totally resides in its libraries, then the replacement libraries mechanism could allow total integration of the legacy system by eliminating its main module.

From a practical point of view, to consider libraries like specialized components is necessary to be sure that we have knowledge skills and tools to make rebuild these elements and make any modification to the software; or else, we must aboard this legacy system as a monolithic one with no separate libraries.

If specializations are partially mixed in the main module of the legacy system, replacement of functions using the new library will limited to those located in the library; nevertheless, some enhanced functionalities in the replacement libraries could be used by the old allowing it to evolve.

### 4.2.4 Transparent Modular Systems Interaction

Another form of interaction can be implemented by developing a common navigation interface for two or more systems. All systems will continue working in a separate way, but users will perceive them as only one system (see Figure 4).

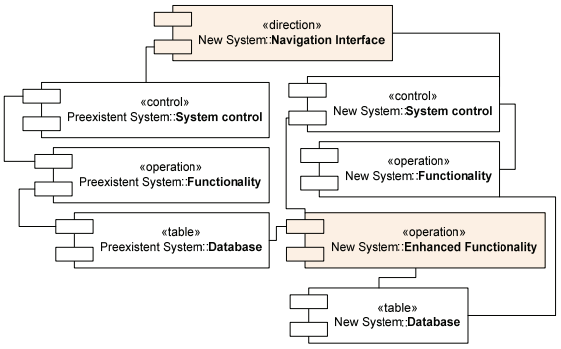


Figure 4: Transparent modular systems interaction.

We have interaction instead of integration, because none of the two systems have conceded partial or total loss of control over their components. A component with enhanced functionality will serve to achieve contextualization, enrichment and exchange of information among both systems.

Mechanism for interaction between two systems can be extended to more systems. Acquired benefits of getting information from a pre-existent system can be extended by adding similar components or sharing this one with more than one system.

### 4.2.5 Evolution by Shared Use of Components with Enhanced Functionality

An enhanced functionality component integrated in the new system can interchange information from each one of the participating systems.

When more than two systems share the same components, as illustrated in Figure 5, we can reduce implementation efforts and enhance usability by introducing standardized behaviour. Functionality enhancement sharing new functionality with all interconnected systems could be considered as a form of evolution.

### 4.2.6 Integration of Systems by Direct Calls to Functions in Operation Components

We can have certain integration between systems that have been developed under compatible, transparent or portable technologies if the new system's control component makes direct calls to functions of the pre-existent system. New system overrides the control component of the pre-existent one and directly executes operation components to use their functionalities (see Figure 6).

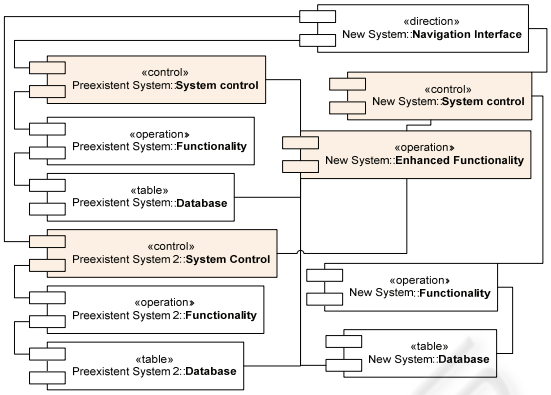


Figure 5: Shared use of an Enhanced Functionality Component.

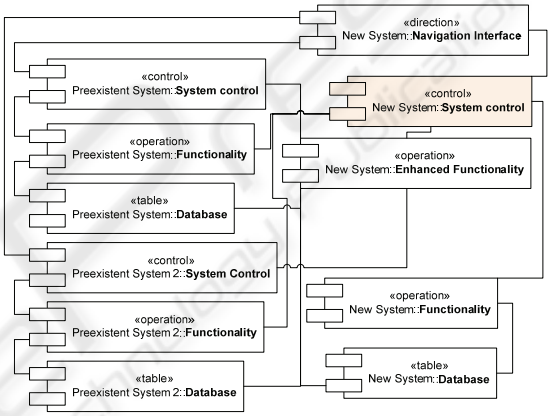


Figure 6: Direct calls to operation components.

This kind of interaction could be useful for critical response systems if it's well managed, but if not, it can cause introduction of inconsistencies.

### 4.2.7 Integration by Coordination of Pre-existent Systems

A kind of partial control yielding from one system to other can be given through coordination. Control component of the new system can make these coordination tasks; and navigation interface can be used to guide execution flows of both systems.

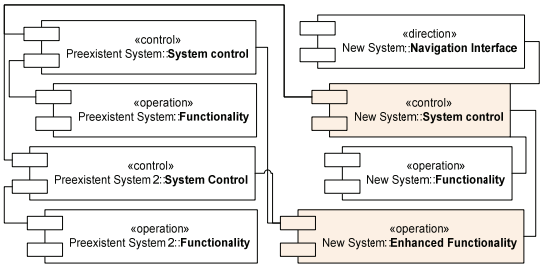


Figure 7: Coordination of pre-existent systems.

As can be seen in Figure 7, calls are made indirectly by making requests to the control component of the pre-existent system. This approach sets a logic that is subject to the decision making process of the new system above decisions taken by the control component in the pre-existent one.

Difference established by a schema of decision taking under the coordination mechanism may seem subtle to the naked eye, but it takes relevance when integrating more than one system. This mechanism becomes common and necessary to ensure that information update movements in different systems is carried out in an appropriate manner avoiding or solving problems of synchronization, integrity, etc.

### 4.2.8 Full Integration and Evolution of Pre-existent Systems

As in the case of legacy systems integration, evolution of a pre-existent system may result in elimination of its control and direction components by replacing them with components from the new system. This approach provides a full integration of the pre-existent system (see Figure 8).

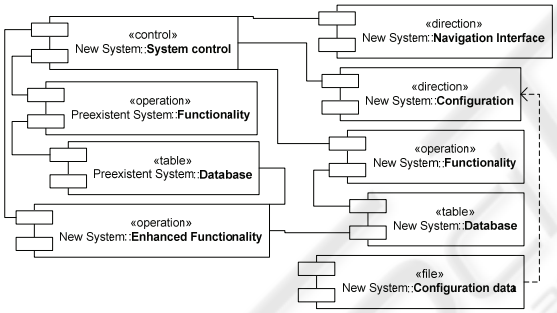


Figure 8: Full integration and evolution of pre-existent systems.

Full integration means that only operation components and information from the pre-existent system will be used in the new system. Evolution will give a path to eliminate the pre-existent system avoiding the risk of letting it become a legacy one.

## 5 CASE STUDY

To exemplify how the set of models could be used to facilitate design of different architectures we have included an example to describe its application.

We describe an application that is part of a nationwide statistical and geographical information network (RNI) defined in a Federal Law in Mexico (DOF, 2008) that is under construction. It must solve statistical information requests from various systems located in different institutions members of the SNIEG (in Spanish for National Statistical and Geographical Information System). The Statistical Information System of SNIEG must integrate information to show results to the user without modifying existent systems.

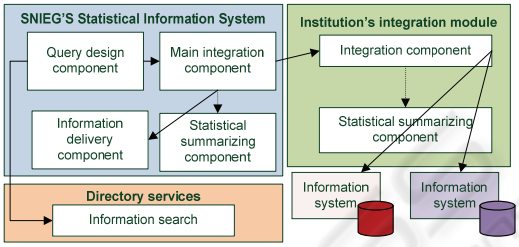


Figure 9: Conceptual diagram of the integration mechanism.

A total summarizing of all information will not be always possible to produce, because it's needed to have a script describing a statistical valid procedure to make this task, if this is not possible, only individual results that have been sent by each institution will be showed. Each institution also needs to integrate information from its own systems to send these results.

As it can be seen in Figure 9, integration must be made in a distributed mode, first each institution will standardize, summarize and structure individual results with extracted data from different information systems; later a main integration component will deal with all individual answers and compose an integrated one to solve user's request.

System will control each institution's integration module by applying the model number seven for "integration by coordination of pre-existent systems", and for calling web services, the sixth model for "Integration of systems by direct calls to functions in operation components" will be developed (See Figure 10).

The component: "RF1. 3 QueriesProcessing" asks for information in each "O1-1 Institution's Integration Module" and composes the full aggregated answer to requests.

With this architectural design, architects solved the problem of integration without modifying any pre-existent system, a critical condition because participating institutions didn't want to do it. Now the mechanism can be implemented in each institution to integrate information and functionality from any statistical system and let the system grow nationwide with a controlled effort.

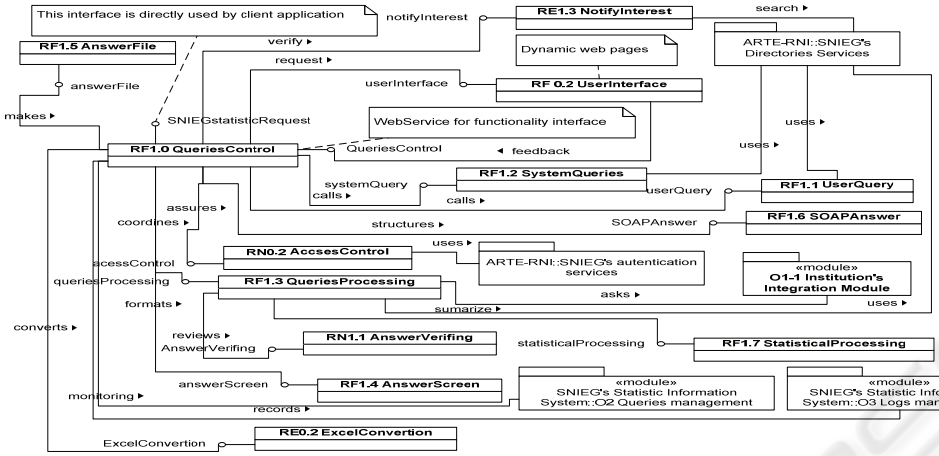


Figure 10: Architectural detail of SNIEG's Statistical Information System.

# 6 CONCLUSIONS AND FUTURE WORK

Direction, control and operation are abstract specialized functions which enables a new approach to system's modularization. These specializations could be the basis for designing architectural structures with enhanced non functional characteristics like: maintenability, interoperability and governability.

Using these abstractions, we have proposed a set of models for systems evolution, interaction and integration that could help to explain, rationality for developing architectural solutions that can be implemented under different technological approaches.

This set could be used as a starting point to classify different kinds of architectures and compare systems under similar contexts.

Also, identification of specialized components, described in this document may be used as a basis for the conversion of architectural patterns into a set of rules that could be applied to develop tools to assist software architecture design tasks.

# REFERENCES

O'Callaghan, A., 2000, Focus issue on legacy information systems and business process change: Migrating large scale systems to component-based and object technology: The evolution of a pattern language, *Communications of the Association for Information Systems, Volume 2, Article 3,* AIS

Buschman, F., Meunier, R., Rohnert, H., Sommerland, P., Stal, M., 1996, *Pattern-Oriented Software Architecture. A System of Patterns. Volumen 1*, Willey, USA

Cornella-Dorda, S., Lewis, G., Place, P., Plakosh, D., Seacord, R., 2001, Incremental Modernizing of Legacy Systems, *CMU/SEI-2001-TN-006,* Software Engineering Institute, Carnegie Mellon University

DOF, 2008, Ley del Sistema Nacional de Información Estadística y Geográfica, Diario Oficial de la Federación 17 de abril de 2008, Secretaría de Gobernación, México

Garland, J., Anthony, R., 2003, *Large-Scale Software Architecture*, Willey, USA

Keshav, R., Gamble, R., Payton, J., Frasier, K., 1999, Architecture Integration Elements, Technical Report *UTULSA-MCS-16-99, Department of Mathematical and Computer Sciences,* University of Tulsa

Muñoz, J. Muñoz, J., Arévalo, C., 2006, Building a Methodology for the Design of Reference Architectures That Integrates Legacy Systems, *IRMA2006*, IDEA Press

Pressman, R. S., 2004, *Software Engineering. A Practitioner's Approach,* McGraw Hill, USA, 6th Edition

Robertson, P., 1997, Integrating Legacy Systems with Modern Corporate Applications, *Communications of the ACM, Vol. 40, No. 5,* ACM Press

Sommerville, I., 2004, *Software Engineering.*, Addison Wesley, USA, 7th Edition

Trowbridge, D., Roxburg, U., Hohpe, G., Manolescu, D., Nadghan, E., 2004, *Information Patterns. Patterns & Practices*, Microsoft Corporation, USA

Ziegler, P., Dittrich, K., 2004, Three Decades of Data Integration – All Problems Solved*?, WCC2004,* University of Zurich