Concern-oriented and Ontology-based Modular Architectural Design of Software Systems

Crenguța BOGDAN
Faculty of Mathematics and Computer Science,
Ovidius University, Constanta, 900527, Romania

ABSTRACT

The design activity is carried out during the development of every software system regardless of the software process model that is used. The design focuses on four detail levels: data structure, architecture, interfaces, and components. In general, these artifacts are constructed by transforming the requirements model obtained in the analysis activity. In this paper, we present an approach to construct software architectures based on the stakeholders' concerns of the information system where the software system will operate. The concerns are analyzed, their related knowledge and beliefs are identified, and a domain ontology is created. Using the ontology in our approach, the software architecture is composed by architectural modules. Each architectural module is constructed using the Model-View-Controller architectural pattern and fulfills additional design rules. We applied the approach in the design and implementation of the software architecture of a system that provides the registration of a new trading company using the services provided by the public administration institutions.

Keywords: design, software architecture, concern, ontology, architectural module

1. INTRODUCTION

The design activity is carried out during the development of every software system regardless of the software process model that is used. The design focuses on four detail levels: data structure, architecture, interface representation, and components. Depending on the detail level, the design activity is decomposed in four sub-activities: data design, architectural design, interface design and component-level design [12]. From these sub-activities, in this paper we focus on the architectural design of software systems.

The architectural design is the activity during which software architecture of the system is constructed. Software architecture is a description of the components that form the software system, their relations to each other that coordinate the actions of these components, and the principles guiding its design and evolution [14]. The components can be modules, objects, web services, and so on, depending on the partitioning criterion used. Using our approach, we will obtain object-oriented software architectures.

The software architectures are useful in the development process of a software system in many ways:

- the software architecture is often the first artifact from design that solves the decisions about how the requirements of the system will be fulfilled or how the stakeholders' concerns will be addressed;
- the architecture is the key artifact necessary to obtain a quality software system. This could be created by reusing of the existent components from the development of the similar previous software systems;
- usually, the architecture is the first artifact read and used by the programmer.

Therefore, the obtainment of a quality software architecture that fulfills the design principles and addresses the stakeholders' concerns is an ongoing problem.

2. BACKGROUND

2.1 Separation of Concerns

The separation of concerns is an old decomposing and composing principle that partitions an information or software system into smaller more manageable and comprehensible parts [11]. Each decomposing criterion is derived from a concern or need belonging to a particular area of interest.

In [2] we defined the (stakeholder's) concern as a problem-originated care of one or more stakeholders involved in the construction or evolution in its natural environment of an information system (IS). The care of a stakeholder derives from his/her interest or responsibility in the IS's real world, his/her thinking to improve or modify something in this world for a better matching of his/her expectations, or worrying whether something wrong or undesired could occur.

The specification of a concern problem uses a pair of two descriptions: the initial state description of the current situation, as the stakeholder perceives it, and the final state description of the situation that matches expectations, interests, or preoccupations of the stakeholder.

These two elements are respectively considered as hypothesis and conclusion of the problem specification. The problem's initial state contains all information and knowledge necessary to obtain the final state of the problem and, thus, to solve it. The high-level specification of a concern that a stakeholder tries to solve is an association of such a pair of states and the role that the stakeholder plays in the system (see the example C5).

2.2 Concerns Analysis

In order to identify the concerns, we start with the analysis of the stakeholders' preoccupations, interests and beliefs, and identify how they generate concerns, in other words how the stakeholders reason. In this respect, we resort to contemporary philosophy that gives us many theories of mind.

The most known and accepted theory is the Functionalism which models the states of mind (beliefs, concerns, desires, needs, being in pain, etc.) by considering solely their functional role: transformers of sensory inputs in behavioral outputs, in causal relations with other states of mind [1]. The states of mind are closely related to the beliefs and knowledge, which are treated in the next subsection.

Beliefs Related to Concerns

We consider a belief as a state of mind about a mental representation that symbolizes a mental object that depends on a perception [16]. In the cognitive psychology, a mental representation is defined as a psychological mechanism that allows the reflection and the knowledge of an entity, phenomenon, or of a state of affairs in its absence. The condition is that, this was previously perceived in the real world [16].

There is a strong relation between knowledge and beliefs: a credible belief accepted by all stakeholders who are interested in, it's a piece of knowledge.

Nevertheless, we do not consider all the beliefs and knowledge of a stakeholder, but only those that belong to the explanations of the cause of problems, which are related to their concerns.

The mental representations of the stakeholders' knowledge and beliefs are formed by concepts that refer to individuals (or instances) belonging to three categories: physical entities and their relations in the real world, ad hoc conceptualizations resulted from the stakeholder's experience, and abstract (non-physical or social) entities that were produced by the human mind and are shared by various communities.

The identification of concepts from every belief and knowledge associated to a concern represents the activity in which a vocabulary is created. The vocabulary is a set of concepts that we use in order to refer to concrete and abstract entities, as well as relations between them from the domains associated with the problems related to the identified concerns. Starting from each belief and knowledge associated to the concerns, the participating concepts are gathered

in vocabulary. The vocabulary is used for solving the problem associated with the concern. This activity is repeated until the whole conceptual domain of the problems associated to the concerns shared between stakeholders is obtained.
Then the foundational ontology is chosen.

2.3 Ontologies

An ontology is a formal specification of the concepts intension and the intensional relations that can exist between concepts. According to Guarino's definition, "an ontology is a logical theory accounting for the intended meaning of a formal vocabulary, i.e. its ontological commitment to a particular conceptualization of the world" [6]. A conceptualization is a set of conceptual (intensional) relations defined on a domain space [6]. Depending on their arity, the conceptual relations are unary ones (and they are called concepts) or binary, ternary and they are called relations.

2.4 Architectural Design

The architectural design is an activity necessary for the construction of a software system. During this activity, the software architecture of the system is constructed.

In general, the software architectures are designed applying an architectural pattern. This describes the kind of components, their relations, their constraints, the design and the composition rules of the components.

In our approach, we use the MVC architectural pattern [3]. This pattern classifies the objects in three categories: model, views, and controller objects. The classification criterion is given by the responsibilities of the objects from each category. View objects are objects with which user stakeholders interact directly, such as frames, forms, panels, and so on. The model objects eventually contain persistent information managed by the system. Many such objects come from the business objects of the domain model of the information system where the software system will operate. However, other objects of this category can also emerge during the design activity of the software architecture. In our approach, the model objects come from the used domain ontology. Finally, the controller objects have the responsibility to manage the logical flow and the events produced by the user stakeholders in their interactions with the view objects.

The rules that constrain the communications between objects are the followings:

- the objects from the same level can communicate among them;
- user stakeholders can access only the view objects;
- view objects can communicate only with the control objects. However, there are cases when the view objects send messages to the model objects, but these messages query their states and do not modify them.
- controller objects can communicate with the view objects;
- model objects communicate only with the control objects.

In order to obtain a quality and modular software architecture the designer must apply the design principles of low coupling, high cohesion, and assignment of responsibilities. These principles are fulfilled if we use the general responsibilities assignment patterns (shortly, GRASP) like Information Expert, Creator, Low Coupling, High Cohesion, Controller, and Polymorphism [8]. Other design patterns can also be used [3].

As modeling language, we use the Unified Modeling Language (UML) which is a standard modeling language used in the analysis and design of the information and software systems [10]. In our approach, we used UML in the construction of the class diagrams and the sequence diagrams of the architectural modules.

3. OUR SOFTWARE ARCHITECTURE DESIGN APPROACH

Having the domain ontology, in this paper we present a design approach of modular software architectures. Each architectural module is constructed using the MVC architectural style and fulfilling additional design rules. These rules are given and explained in what that follows.

The first rule is that we construct an architectural module for each concern. In this way, we are sure that every concern is taken into consideration during the design phase. The architectural modules can also be constructed in a certain order. For instance, the order could be given by the dependency graph of the concerns. However, the way in which we construct such graph will not be treated in this paper.

Furthermore, using our approach each concern has associated a controller object. Every concern can be addressed and we obtain a better assignment of responsibilities in classes. The controller object responsibilities are two-dimensional: a) managing the events produced by users and sending of their requests to the model objects, and b) transmitting the results to show them in the graphical interfaces.

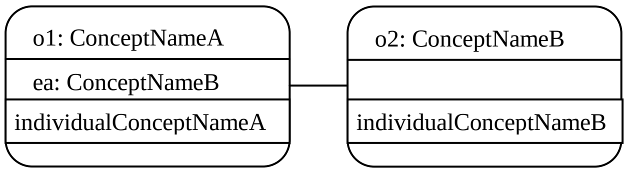
Information from the belief or knowledge associated with a concern gives us an idea about the individuals of the concepts, which will be used in an architectural module. Thus, for each concept belonging to a belief or knowledge we decide if we need to instantiate it and the created individual is used by the software system. In the affirmative case, we add to the architectural module a model class that maps the concept from the used ontology. Otherwise, we deal with one of the three cases:

- we already have in the architectural module a model class that maps the concept or a related one and we used it if this is linked by an ontological relation (or property) to another class;
- within the ontology we seek a related concept, we create a model class for it adding, and we add it to the architectural module; or
- we ignore it because the system does not use it.

In addition, each object of a model class is a wrapper of an individual of the mapped concept, i.e. the object encapsulates an individual of the concept and can contain extrinsic attributes.

These attributes emerge from the fact that the object is linked by other ones with which it collaborates. The type of an extrinsic attribute is the class of the object that participates in the collaboration. This rule can be graphically presented like in Fig. 1. Through the extrinsic attribute of the o1 object, we can use the individual contained by the object o2, if the later gives public access.

Fig. 1. The object model of two objects, which



contain individuals

Another kind of model objects is constituted by the composite objects. These contain model objects created at the beginning or during the system execution. The idea is that if there are many objects of the same class and they have to be used during the system execution, we temporary put them in a composite object that is linked using aggregation by the object parts. In addition, a composite object has the responsibilities to manage the creation and the state modifying the objects parts. In this way, we also fulfill the Creator pattern [8].

Furthermore, the controller objects might send requests to the composite object that solves them. These objects are useful especially when they have to be unique during the system execution. In these cases, we apply the Singleton pattern [3] to the composite objects. We graphically present these ideas in the class diagram from Fig. 2, where we have a unique composite object of the ConceptNameAggregate class that contains a collection of objects of the ConceptName class. We observe in Fig. 2 that the ConceptNameAggregate class fulfills the Singleton pattern containing three elements:

- a single and private constructor (the operation create());
- a private and static variable named instance of the type ConceptNameAggregate, i.e. the class where it belongs, and
- a public and static operation called getInstance() that returns (using the variable instance) a reference to the unique object of the class.

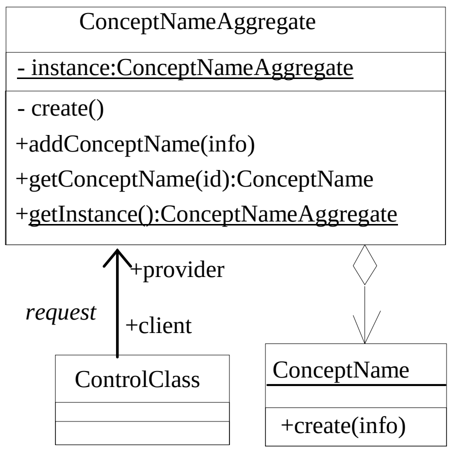


Fig. 2. A class diagram in which three design patterns are applied: Singleton, Creator, and Low Coupling

We also show the sequence diagram (Fig. 3) of the collaborating objects when a controller object creates an object of the ConceptName class. In this figure, we observe that we applied the Creator and the Low Coupling design patterns [9] by assigning the object of the ConceptNameAggregate class to the responsibility to create objects from the ConceptName class (Creator pattern) and the controller object sends all necessary information to the composite object (Low Coupling pattern).

Composite objects could also come from the ontology, not only from a design decision. In other words, if an A concept from a belief or knowledge is linked to another B concept by the temporal and temporary parthood or the constitution relation, then we add the B concept to the architectural module as a model class. After that, we link the B class to the class that maps the A concept by the UML aggregation relation. Finally, we verify if the aggregate class has to manage exclusively its parts. In the affirmative case, the aggregation relation becomes a composition one.

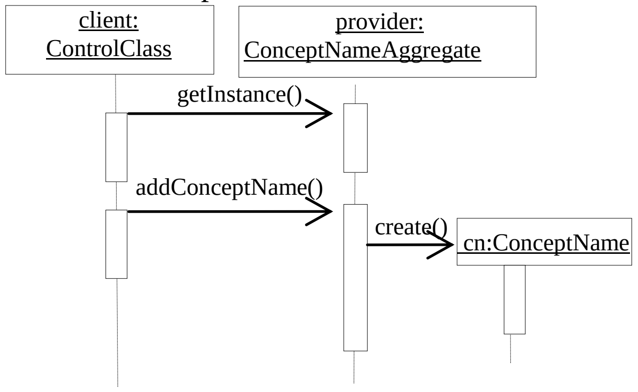


Fig. 3. The sequence diagram of the scenario in which a composite object creates an object part with the information received from a control object

The last category of the model objects is constituted by the manager objects. These objects have the responsibility to manage the operations with the ontology. The manager objects deal with two kinds of operations: loading and saving the individuals from/in ontology.

The manager objects execute the saving operations so that the ontology consistency is maintained. This responsibility raises problems due to the restrictions imposed by ontology. These problems can be solved in the design or implementation phase of the software system. As we decided to use the Jena framework [7] to work with the ontology, we chose to solve the restriction problems in the implementation phase. The idea is that before an individual of a concept is saved in the ontology, we verify if it is linked by properties with restrictions by other individuals. In the affirmative case, the manager object sends the properties and range classes to the controller object (that manages the concern), and these classes have to be instantiated. The controller object creates a view object that could be a form or a graphical interface and show it to the user that will fill it with the necessary information.

The view level of an architectural module is constructed according to the design rule: any knowledge or belief that needs information from the user has an associated view object. This could be applied in one of two cases: a concept from a knowledge or belief does not have individuals in the ontology or the associated class from the architectural module does not have objects necessary for the system functioning, or we are not in the previous case, but we need confirmations from the user.

4. CASE STUDY

Our approach is applied to the information system of the Romanian Public Administration (short, RPA). The RPA includes public institutions like the TRO, the Public Finance Administration, the Labor Safety and Social

Insurances Agency, the Official Gazette Agency, etc..

The RPA also provides services to companies belonging to the business environment or to private entrepreneurs who want to establish their own company. In order to do this, the founder has to register his or her company at the TRO in the city where the company is based. This public institution issues a registration certificate that authorizes the legal operation of his or her company.

4.1 Identification of Concerns and their Related Knowledge and Beliefs

For the first step of the approach, we identified the stakeholders who have a legitimate interest in the information systems considered. They are applicants such as founders, administrators, legal representatives, including companies or corporate entities, and clerks, jurists, judges, operators of service providers such as public institutions and banks.

For the second step, we identified the concerns of the stakeholders, more precisely 31 concerns of the founder and the other stakeholders. Below, we give the description of a founder's concern.

C5	Name: Care to state the new trading company's name	
	Problem	Hypothesis: The founder has to choose at least three Romanian names. These names will be verified by the TRO. According to art. 39 Law no. 26/1990 regarding the Trade Register, the names cannot contain certain words. Conclusion: What name will the new trading company have?
	Stakeholders: Founder	

The next two steps consist in the analysis of the concerns and business rules. The aim of the analysis is the identification of the pieces of knowledge and beliefs. For instance, in the table below we provide two samples of knowledge and belief of the concern C5.

Table 2. Mental representation descriptions of the beliefs and knowledge of C5

Code	**Mental representation description in natural language**
B10	Every trading company name may contain the words: "national", "Romanian", "institution" or their derivates subject to the consent of the Government General Secretariat.
K5	All the trading company names are reserved by the TRO.

As statistical information from the concerns analysis we derived 204 beliefs and pieces of knowledge and from the business rules we obtained 60 beliefs and pieces of knowledge.

4.2 Our Ontology

Furthermore, from each belief and piece of knowledge we identified the concepts and their conceptual relations. Then, we analyzed them and, using the DOLCE [9] and D&S [4] ontologies, we described the intension of the concepts and their conceptual relations.

In DOLCE, the restrictions are given using a subset of the first-order logic and their verification is a long time task. That is why we translated our domain ontology in OWL DL (Web Ontology Language-Description Logic) language [15] and we checked the ontology consistency with the help of the Protégé tool [5] and the RacerPro reasoner system [13]. In Fig. 4 we show an excerpt of our ontology in the OWL language.

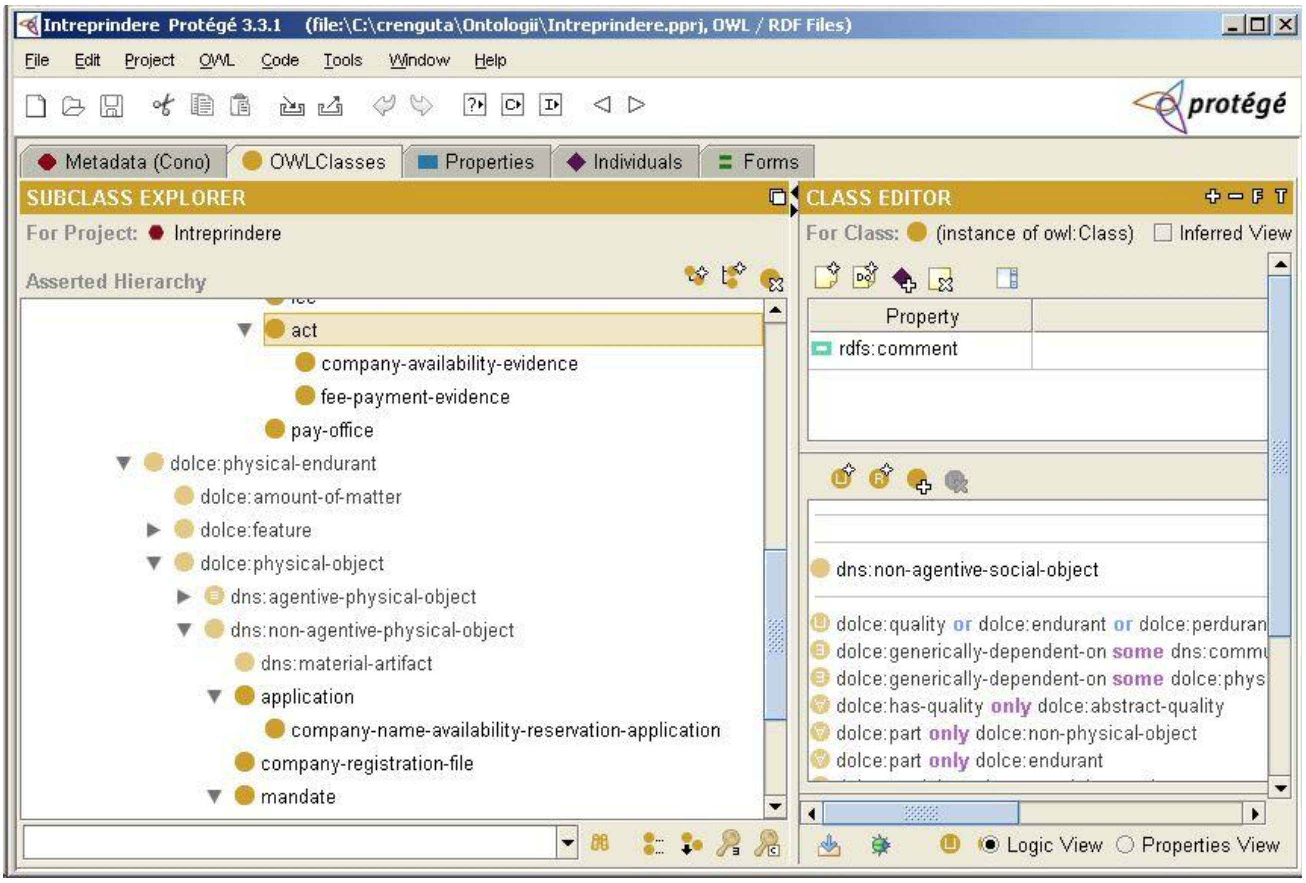
4.3 The Application of our Design Approach

Based on the design approach described in Section 3 we constructed the software architecture of a software system that provides the registration of a new trading company using the services provided by the public administration institutions.

The software architecture is formed by 31 architectural modules, one module for each concern. The architectural module associated to the concern C5 is presented in Fig. 5. Based on limited space reasons, we present only the architectural module of the concern C5 classes and their relations, but the classes do not contain attributes and operations.

Each architectural module is constructed using the domain ontology and fulfilling the design rules of our approach.

We started to implement the software system using the Java language. In order to use the domain ontology, we use the Jena framework [7] also coded in Java. Due to the Jena framework,


Intreprindere Protégé 3.3.1 (file:\C:\crenguta\Ontologii\Intreprindere.pprj, OWL / RDF Files)
File Edit Project OWL Code Tools Window Help
protégé
Metadata (Cono) OWLClasses Properties Individuals Forms
SUBCLASS EXPLORER
For Project: Intreprindere
Asserted Hierarchy
CLASS EDITOR
For Class: (instance of owl:Class)
Inferred View
Property
Logic View Properties View

act
company-availability-evidence
fee-payment-evidence
pay-office
dolce:physical-endurant
dolce:amount-of-matter
dolce:feature
dolce:physical-object
dns:agentive-physical-object
dns:non-agentive-physical-object
dns:material-artifact
application
company-name-availability-reservation-application
company-registration-file
mandate

rdfs:comment

dns:non-agentive-social-object
dolce:quality or dolce:endurant or dolce:perduran
dolce:generically-dependent-on some dns:commu
dolce:generically-dependent-on some dolce:phys
dolce:has-quality only dolce:abstract-quality
dolce:part only dolce:non-physical-object
dolce:part only dolce:endurant

Fig. 4. Excerpt of our ontology in the OWL language

we added to our software architecture two classes: Ontology and MyDBConnection. The first class is used in order to create a Jena persistent model that maps the used ontology. The second class gets the connection with the used MySQL database.

5. CONCLUSIONS

In this paper, we presented a concern-oriented and ontology-based approach to design quality modular software systems divided in architectural module.

Each architectural module is associated to a concern and uses classes that map concepts from the used domain ontology. From the technical point of view, an architectural module is constructed using the MVC architectural pattern, design patterns, and additional design rules.

The advantages of our approach are: a) as a concern-oriented approach, it allows the system designer to solve the concerns (i.e. to solve the problems associated to concerns) step by step, with the help of the system stakeholders. In this way, the solving of a concern's associated problem can be decoupled by the solving of other problems; b) the architectural module associated to a concern could be used in other software architectures for other sub-processes of the public administration domain, if we have the same concern.

However, the approach has a limitation in the sense that it is not a universal one, more precisely it can be applied for the developing of the 3-tier software architectures that provide to users one or more graphical user interfaces.

6. ACKNOWLEDGEMENTS

Special thanks to Coralia Martinescu and Eugen Dumitru for their support in the ontology construction and software system development.

7.REFERENCES

[1] N. Block, **Readings in Philosophy of Psychology**. Cambridge, Harvard, 1980.
[2] C. Bogdan, L.D. Serbanati, "Toward a Concern-Oriented Analysis Method for Enterprise Information Systems", **IEEE International Multi-Conference on Computing in the Global Information**

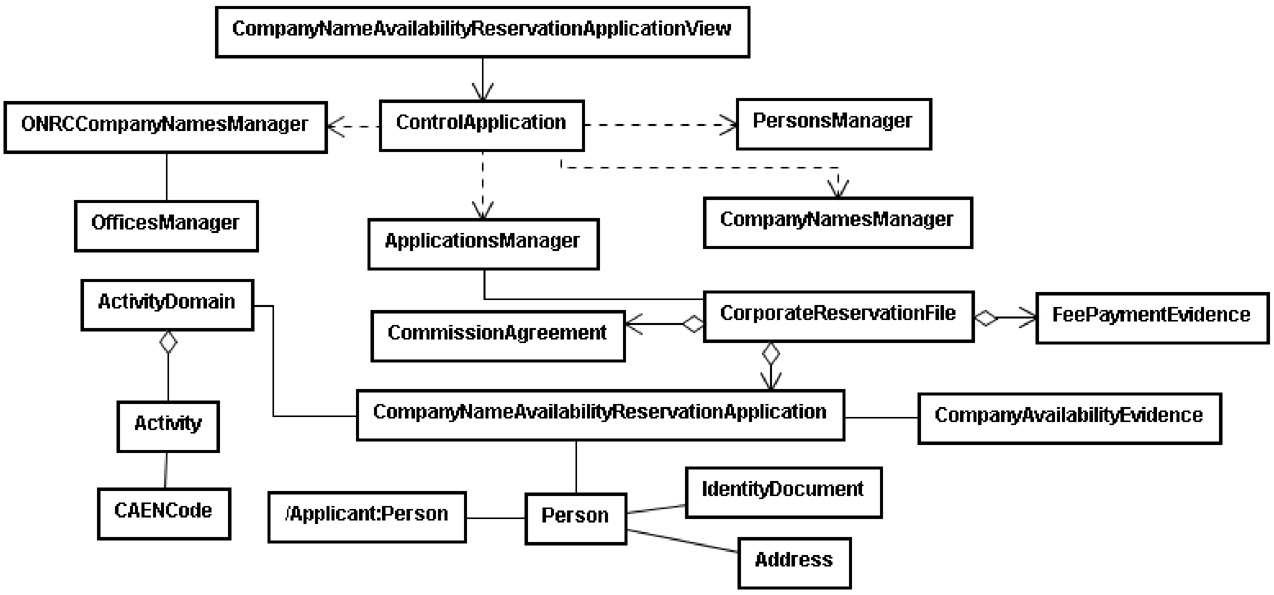


Fig. 5. The architectural module of the concern C5

Technology (ICCGI 2006), Bucharest, Romania, 2006.
[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison Wesley Professional, 1994.
[4] A. Gangemi, P. Mika, "Understanding the Semantic Web through Descriptions and Situations", **Proceedings of the International Conference ODBASE03**, Italy, Springer, 2003.
[5] J. Gennari, M. Musen, R. Fergerson, W. Grosso, M. Crubzy, H. Eriksson, N. Noy, S. Tu, "The evolution of Protégé-2000: An environment for knowledge-based systems development", **International Journal of Human-Computer Studies**, 58(1), 2003.
[6] N. Guarino, "Formal Ontology and Information System", **Proceedings of FOIS'98**, Trento, Italy, IOS Press, 1998.
[7] Jena Framework, link: http://jena.sourceforge.net/, 2007.
[8] C. Larman, **Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and the Unified Process**, Prentice Hall, 2004.
[9] C. Masolo, S. Borgo, A. Gangemi, N. Guarino, A. Oltramari, "WonderWeb Deliverable D18. Ontology Library", **IST Project 2001-33052 WonderWeb: Ontology Infrastructure for the Semantic Web** (2003)
[10] OMG, **Unified Modeling Language Superstructure**, version 2.0, ptc/03-0802, 2003.
[11] D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules", **Communications of the ACM**, 15(12), 1972.
[12] R. S. Pressman, **Software Engineering. A Practitioner's Approach,** McGraw-Hill Publishing Company, 2000.
[13] RacerPro Reasoner, http://www.racer-systems.com/, 2008.
[14] The Institute of Electrical and Electronics Engineers (IEEE) Standards Board, **Recommended Practice for Architectural Description of Software-Intensive Systems** (IEEE-Std-1471-2000), 2000.
[15] World Wide Web Consortium, **OWL Web Ontology Language Reference**, W3C Recommendation 10 February 2004, http://www.w3.org/TR/2004/REC-owl-ref-20040210/, 2004.
[16] M. Zlate, **Cognitive Mechanisms Psychology**, Polirom, 2004.