reviewed paper

Mobile Museum Guides Applications based on Knowledge Graphs

Dmitry Zamula, Dmitry Muromtsev, Nataly Zhukova

(Dmitry Zamula, ITMO University, Saint Petersburg, Russia, zamula.dmitry@gmail.com)
(Dmitry Muromtsev, ITMO University, Saint Petersburg, Russia, mouromtsev@mail.ifmo.ru)
(Nataly Zhukova, ITMO University, Saint Petersburg, Russia, nazhukova@mail.ru)

1 ABSTRACT

In the paper, we discuss our experience in design and development of a content consumption focused mobile applications with data sources in the form of Linked Data by the example of developing museum guide application for The State Russian Museum. We describe our approach to formalizing a model in a dynamic-typed programming language (JavaScript) and the way to keep it consistent. The paper contains the description of the system's main component: a framework for generating a model from an ontology. Ontology-based application architecture can facilitate Domain Driven Design approach, and we demonstrate the ways how to combine these techniques in practice. Lastly, we discuss challenges and problems we faced during the development, then present our conclusions and future direction for exploration.

Keywords: museum guide, knowledge graphs, mobile application, programming, framework

2 INTRODUCTION

Today, a usage of Linked Data and knowledge graphs could effectively solve the challenges of interaction with multilingual, heterogeneous, interlinked data [1]. The typical goals of using a Knowledge Graph are domain specific information collecting and simplifying of usage that data. From software development perspective it expresses in requirements for unifying data structure, existence the ability to update and enrich data without breaking the old structure, and simplifying a program's business logic in part of the interaction with data sources. In most cases, this approach is used to create Web applications [2], but there is a lack of similar solutions for mobile platforms. This article describes the experience of making a mobile application based on the Semantic Web technologies.

3 REQUIREMENTS AND USE CASES

The goal of our project was to implement a mobile assistant for museum visitors. The app intended for The State Russian Museum, but later could be adapted to other similar institutions. The main scenario for visitors was to open a general page about selected art object and start browsing all related information entities in "Wikipedia" way, without strict pages order. The main difference from similar mobile guides was the use of the hypertext principle for navigation in the application with the ability to create explicit UI structure for specific pages if needed. In the process of reading an article about a painting, a user could move to an author's page, where painter personal information was presented in a structured way with a link to the page catalog of all his works, etc. It was necessary to implement this application for multiple mobile platforms (Android, iOS).

At the start of the project, the backend side was already implemented as a web application based on a semantic content management system "Metaphacts". All data were stored in high-performance graph database Blazegraph in RDF format. There was a SPARQL endpoint as an external interface for this data.

4 BASE TECHNOLOGY STACK

At the stage of selecting the base technology stack for developing the application, we made a comparison of two main approaches in the field of mobile app development: the creation of native applications by using the tools provided by the developer of the platform, or the implementation of cross-platform applications, using third-party technology stacks. One of the popular solutions for cross-platform development is an approach to building a hybrid application [3]. The main part of such app is a Web page that contains a user interface (built on HTML + CSS) with business logic in JavaScript, and the container which acts as a wrapper of that web page, and implements the interaction with the mobile platform API, allowing developers to use functionality of the mobile operating system, which has no direct mapping in a standardized JavaScript API (for example, determining the orientation of the device in space can be accomplished by DeviceOrientation events from DOM API, but an interaction with the file system requires a non-standard API, which is provided by container).

The main advantage of this approach is the possibility to reuse much of the code for different operating systems. You can often find references to the problem of the visual aspect of such applications as UI is not consistent with the recommendations of the particular operating system and, accordingly, the main platform style. To date, this problem can be considered as solved, since there is a lot of frameworks in that area for defining the platform type in an application startup time, and rebuilding the user interface following the recommendations for a particular platform (e.g. http://ionicframework.com/, http://www.idangero.us/framework7, https://www.sencha.com/products/touch/).

Also, there is a large community in that area which implies a big amount of third-party components (https://onsen.io/, http://www.telerik.com/kendo-ui, http://mobileangularui.com/). The main disadvantage in hybrid approach is a poor performance of such applications, which is especially noticeable in the heavy, complex user interfaces.

In our case, with the lack of heavy computing tasks in the application as well as the relatively simple user interface, it was decided to use the approach of a hybrid application, due to the resulting advantages, compared to the native applications, especially the code reuse for different mobile platforms. That approach necessitates the use of the JavaScript programming language for describing business logic.

5 DOMAIN MODEL IMPLEMENTATION

Each software in one or another form describes a model as an abstraction of the real world in the form of program code [4]. The part of our project's model is presented on Figure 1.

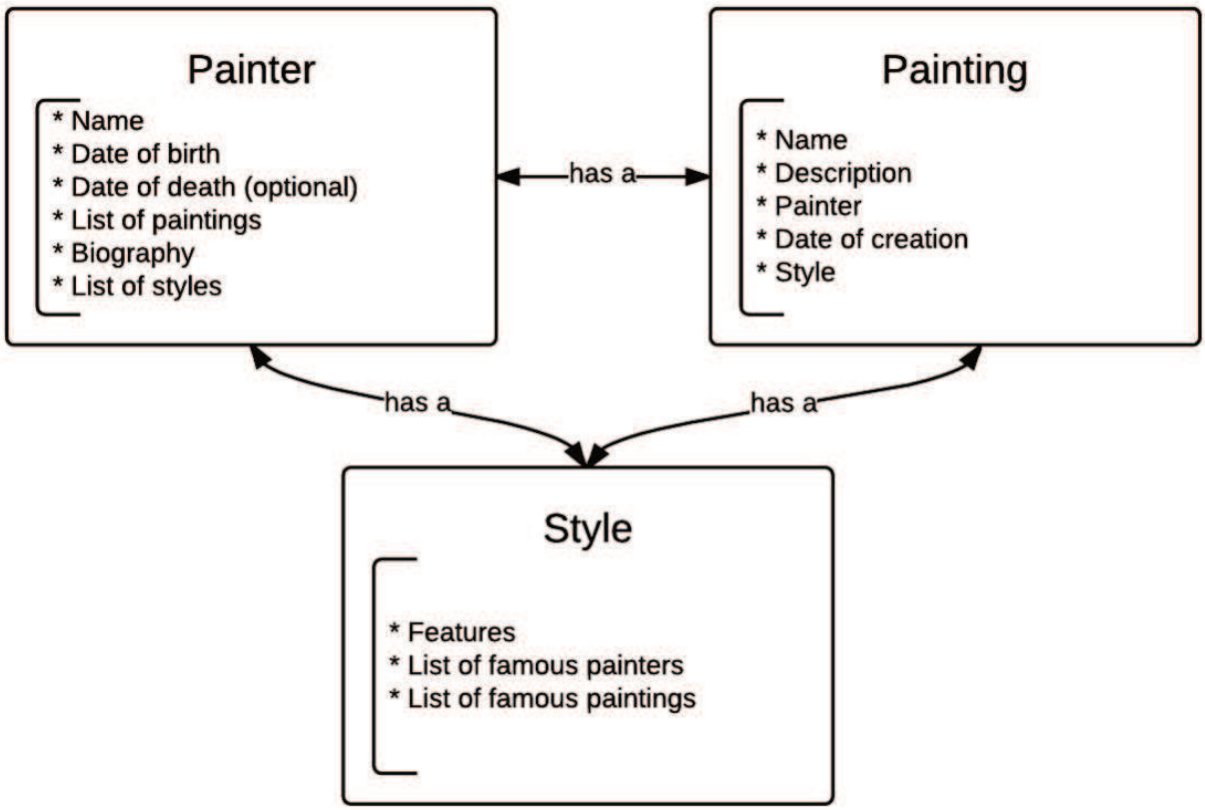

Figure 1. The part of paintings domain model.

The model in the current enterprise programming paradigms (procedural, object-oriented) is a single layer of the application which contains a description of the basic entities and their relationships (inheritance and composition), types of the properties, and interaction interfaces. In the classic 3-tier software architecture [5], the bulk of the business logic is implemented in the form of services that operate on elements of the model. With that approach, the model is represented as a mapping of entities and their properties to the program structure, without methods for data transformations. This method is described by the term "anemic model" [6] and initially was interpreted as an anti-pattern. However, in the process of methodologies evolving, this

approach was reconsidered, so now it is widely used in industrial programming [7], due to the following features that make it easier to build software:

- Explicit separation the data from the processing
- Simplification of building programs that do not have state (stateless)
- Possibility to implement transparent bindings between code and databases (ORM, ODM, etc.)
- Simplification of writing unit tests due to reducing the number of mocks (instead of mocking data objects we can just create and populate them with their constructors as we don't have additional logic inside)

The model layer can be designed by the programmer manually, or generated from some external representation. There are techniques and tools of model generating that based on the structure of a relational database (Hibernate Reverse Engineering toolset), Web service interfaces (wsdl2java), modeling language (Eclipse UML to Java Generator). In our project, there was an API in the form of SPARQL endpoint that allowed us to create external queries for data in the semantic form. The semantic representation of data is an approach for handling of information with the formal description of its inner structure, based on the relationships of the real world objects. In Semantic Web technology stack this approach is implemented by exposing the statements in form "subject property object" that describe the data, its structure and relationships by using specialized tools - RDF data format, SPARQL query language and tools for describing ontologies (e.g. OWL language), which are the formalization of data models in the context of specific area. Ontologies as a technology for data scheme representation was created to unify the description of the structures of information by introducing opportunities for the reconciliation of the definitions of entities from different domains and abstraction to describe the knowledge. OWL language can be represented in RDF format [8], which eliminates the need for a specific set of tools to interact with them, so we can use SPARQL for ontology introspection.

Since the main purpose of ontology is a description of the data structures and relationships in any subject area, we wanted to reuse this description to generate an application model in a source code. Our SPARQL endpoint allowed us not only to work with the data, but to use the existing ontology to describe the application model.

6 ONTOLOGY-BASED MODEL GENERATION

To date, there are several projects that translate the ontologies in the software model [9][10]. Most of these projects support only Java language, and work as a source code generators before the compilation of a program.

In our project, we proposed an analogue of this method for dynamically typed language JavaScript. The solution was to extract information about the model at the runtime, and save it in an internal object which represents the formal description of the domain knowledge based on the types and the relationships between them. A part of the generated model is shown on Figure 2.

This model allows verifying data types of variables that are used on the service level in business logic.

We have developed a framework for generation domain models from ontologies in the source code and for supporting its consistency (data types and relationships between classes) during the execution of the program.

Project "Object Model" [11] was used as a tool for building model definitions in JavaScript and performing type checking at runtime. The main idea can be illustrated by this example:

When we declare a model definition:

```
var User = new Model ({
    name: String,
    female: Boolean,
    birth: Date
 });
```


ISBN 978-3-9504173-2-6 (CD), 978-3-9504173-3-3 (print)
Editors: Manfred SCHRENK, Vasily V. POPOVICH, Peter ZEILE, Pietro ELISEI, Clemens BEYER

the library ensures that all three object properties (name, female, birth) will contain only data that is compatible with the defined types. In the case of violation this conditions, such as assigning to the variable “birth” a numeric value with type Number, an exception will be thrown:

```
var joe = new User ({
    name: "Joe",
    female: false,
    birth: 1986
 });
```

TypeError: expecting birth to be Date, got Number 1986

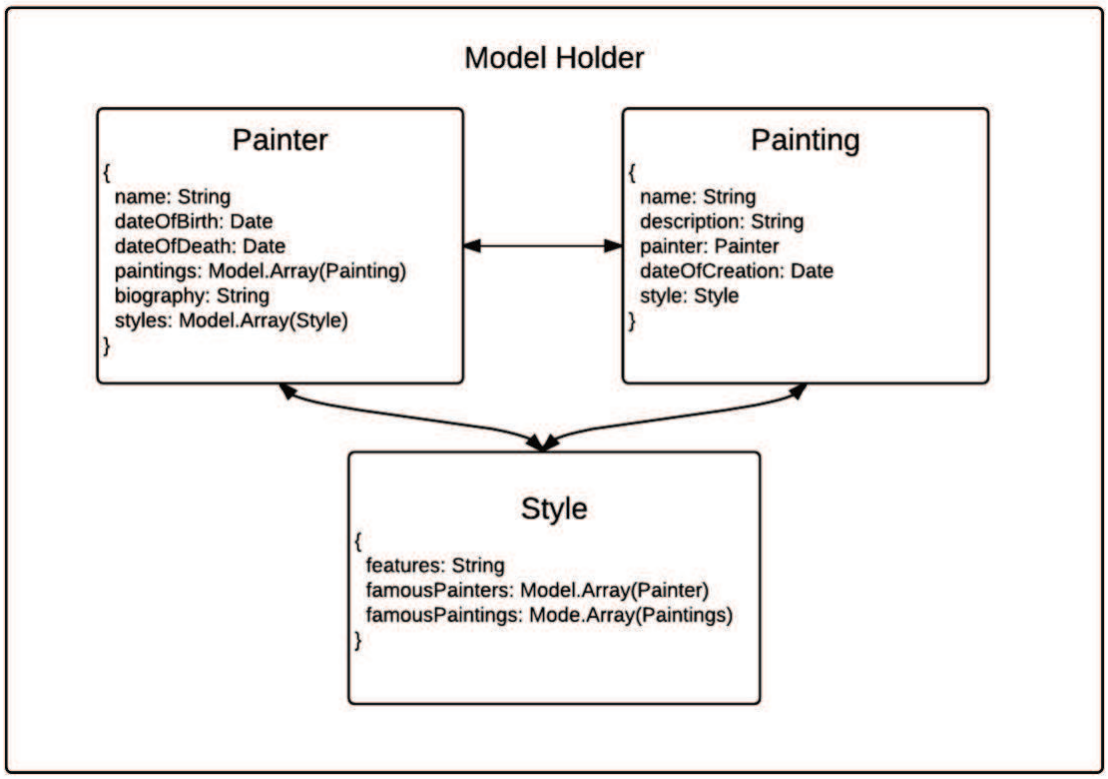

Figure 2. The part of generated model with types and relationships between them.

By using this approach to formalizing the model we were able to automatically generate specific data types for our application from the given ontology. As a technology for working with ontology, we have used SPARQL. In the form of pseudocode, the process of model building can be presented as follows:

```
var models = {}                                                          (1)
var prefixes = [list of URI prefixes]                                    (2)
var ontology = loadOntology()                                            (3)
var ontologyInRdf = convertOwlToRdf(ontology)                            (4)
var store = createLocalRdfStore(ontologyInRdf)                           (5)
var results = store.execute(“ SELECT DISTINCT * WHERE {s a owl:? Class} ”, ...)   (6)
removePrefixes(results, prefixes)                                        (7)
for result in results:                                                   (8)
 models[result.className] = new Model()                                  (9)
 for property in result.properties                                       (10)
   models[result.className].definition[property.name] = property.type    (11)
```

We create a model holder object (line 1), define the list of prefixes for abbreviating generated classes and properties names (2). After loading an ontology (3), we convert it to RDF representation and create a local SPARQL endpoint on that ontology using rdfstore-js library [12] (4). By executing SPARQL queries, we obtain information about classes, properties, and types (line 6). We modify extracted data by removing prefixes from identifiers for readability purpose (7) and after that generate a model by specifying the classes, properties, types and relations with Object Models' helpers (by using constructor "Model").

There is an example of SPARQL query for extracting classes' names ("SELECT DISTINCT * WHERE {s a owl:? Class}"). For obtaining properties' names we use similar query:

SELECT DISTINCT ?prop WHERE {prop rdfs:? Domain ecrm: ${className}. }

where ${className} is a placeholder for a specific class name.

The main system's components are presented in Figure 3. In that architecture we distinguish the following components:

- OWL Syntax Converter for transforming ontology definition in RDF format
- Local SPARQL endpoint for querying the ontologies (based on rdfstore-js library)
- List of SPARQL queries for extracting specific information about classes, properties, and related data
- Object Model, as a library which helps us to formalize a model in JavaScript - the result of ontology introspection.

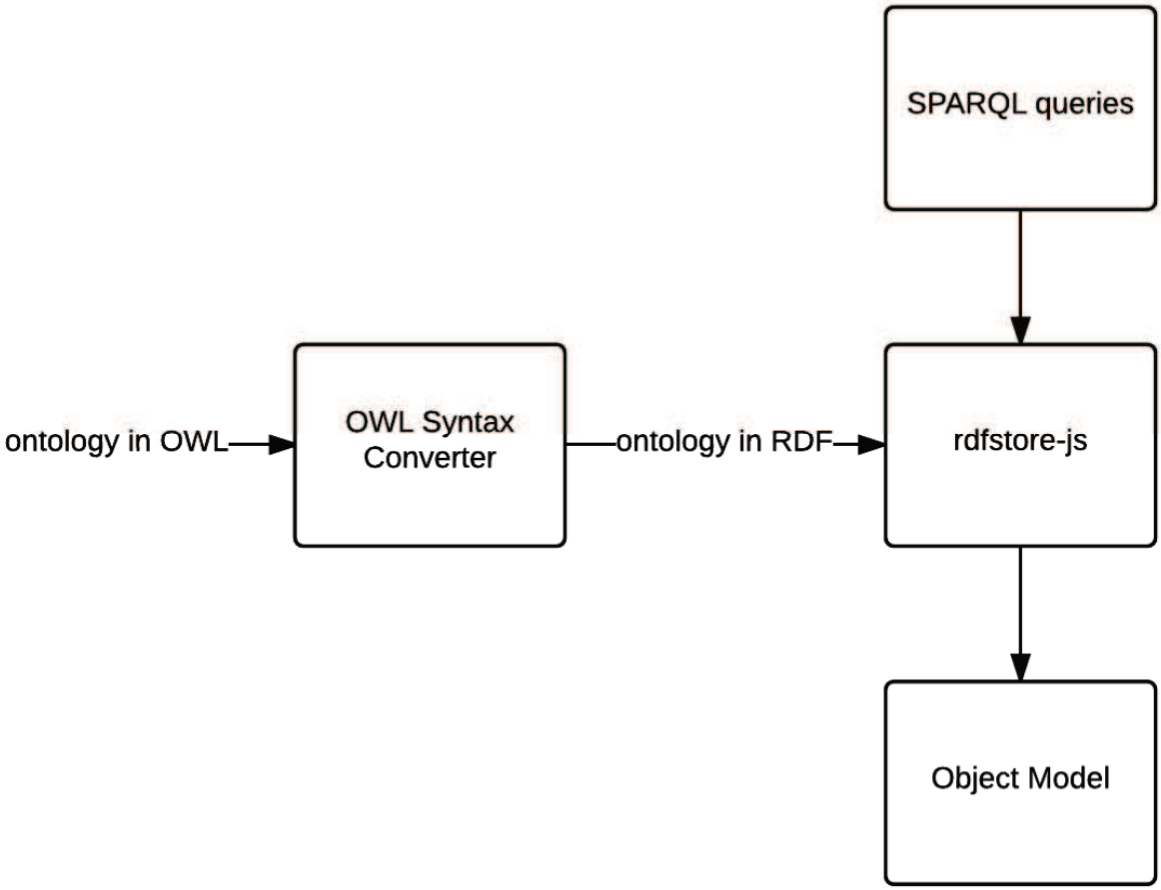

Figure 3. Main components of the framework.

For the current state of the framework, we mark all of the properties as optional, so every class in the model could be created without setting a particular set of fields. The task of filling the required properties is carried out directly by programmers.

ISBN 978-3-9504173-2-6 (CD), 978-3-9504173-3-3 (print)
Editors: Manfred SCHRENK, Vasily V. POPOVICH, Peter ZEILE, Pietro ELISEI, Clemens BEYER

Using the presented approach, we were able to define classes with the properties and their types, as well as to describe the inheritance between classes (at this stage, as an experimental add-on, we allow to use a multiple inheritance).

With the object that keeps information about the application model, we were able to simplify the usage of domain driven design principles in the working process, even with anemic application model. We standardize a common glossary of terms in the application (introducing the ubiquitous language) as there is the main description of the subject area in a separate layer (and, it is assumed that the editing of that layer will be available only for domain experts and application architects). That approach solves the problem of synonyms when one domain object is described by the different classes, which implies confusion and complicates reuse of existing code. Using the feature of ontologies interoperability, we were able to implement the concept of "Bounded Context" - to separate parts of the model in accordance with the various subject areas by using the different ontologies. As an example of such context we can use an ontology for describing a specific part of platform API (e.g., ontology geolocation chamber, NFC, etc.).

Thus, we have the domain model, which can be used at the service level for building business logic. In our project, we completed the model with additional entities related to the technical structure of the project by implementing an additional model object which was formed by programmers. We didn't use the approach of formalizing this part of the model in the form of a specialized ontology due to the small number of "technical" objects in the model, and their relationship with elements of external frameworks (for example, work with the built-in services and AngularJS templates).

As an example of the model's usage on the services layer there is a real method for displaying information on a given painting:

```
function showPaintingsInfo (E22_Man_Made_Object) {
  E22_Man_Made_Object.label = capitalizeFirstLetter (E22_Man_Made_Object.label);
  $scope.painting = E22_Man_Made_Object;
  // ...
}
```

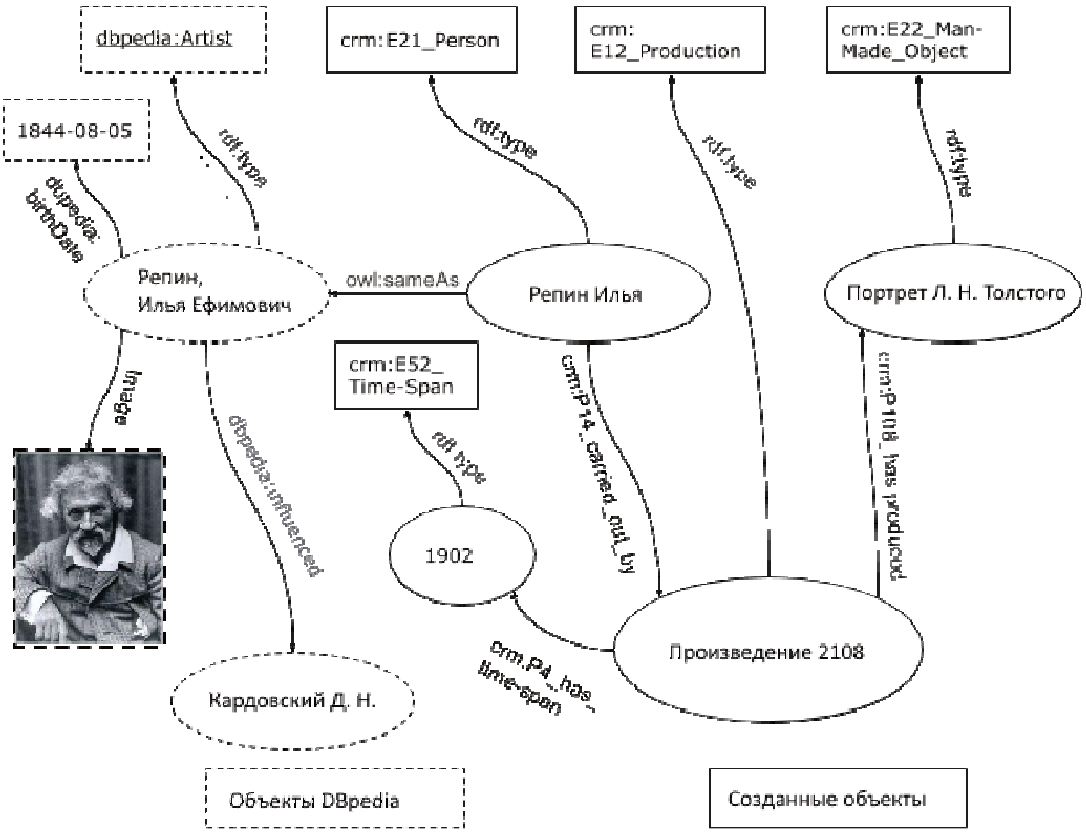

Figure 4. Properties of model's element E22_Man-Made_Object, involved in the logic of painting's description.

7 EVALUATION

In this method, in addition to simple logic for modifying the names of the picture, there is a transfer of the object model into an object $scope, which acts as a buffer between UI service layer and UI templates, so our model can be accessed from the template of application's user interface.

The part of a generated model of our application is presented in Figure 4.

As a result, this part of the model is shown in the user interface (Figure 5):

Figure 5. The final presentation of the model's objects in a user interface.

The ontology-based model allows us to simplify the adaptation of the final application to the needs of other museums. With the unified data structure which is based on the relationship of real world objects, the only significant change, in this case, may be a modification of the user interface templates.

This approach allows reduce the time of development a software product, due to the automation of model layer building phase, as well as simplify the future application support.

8 CONCLUSION

In the process of implementation a mobile application for providing additional information to museum visitors, we have developed a framework for generating a software model from a given domain ontology with dynamically-typed language JavaScript. This tool has been tested in a real project, with the following positive results:

- Reducing costs for the development and modification of the software layer that forms the domain model
- Providing a single domain language for developers
- The formalization of the domain model in a dynamic-typed language

ISBN 978-3-9504173-2-6 (CD), 978-3-9504173-3-3 (print)
Editors: Manfred SCHRENK, Vasily V. POPOVICH, Peter ZEILE, Pietro ELISEI, Clemens BEYER

This approach, however, requires further study, particularly from methodological view, due to the introduction of ontologies in a software development process. The main effect to the process, in this case, brings the prohibition of explicit modification of the domain layer by the programmers and the creation of a separate group inside the development team, specifically for creating and maintaining ontologies.

9 REFERENCES

[1] E. Hyvönen, "Publishing and Using Cultural Heritage Linked Data on the Semantic Web," Synthesis Lectures on the Semantic Web: Theory and Technology, vol. 2, no. 1, pp. 1–159, 2012.
[2] A. Sheth, Semantic services, interoperability, and web applications: emerging concepts. Hershey, PA: Information Science Reference, 2011.
[3] H. Heitkötter, S. Hanschke, and T. A. Majchrzak, "Evaluating CrossPlatform Development Approaches for Mobile Applications," Lecture Notes in Business Information Processing Web Information Systems and Technologies, pp. 120–138, 2013.
[4] B. Selic, "The pragmatics of model-driven development," IEEE Softw. IEEE Software, vol. 20, no. 5, pp. 19–25, 2003.
[5] R. Heckel, R. Correia, C. Matos, M. El-Ramly, G. Koutsoukos, and L. Andrade, "Architectural Transformations: From Legacy to Three-Tier and Services," Software Evolution, pp. 139–170, 2008.
[6] M. Fowler, "AnemicDomainModel", http://www.martinfowler.com/bliki/anemicdomainmodel.html.
[7] "The Anaemic Domain Model is no Anti-Pattern, it's a SOLID design", SAPM Course Blog. https://blog.inf.ed.ac.uk/sapm/2014/02/04/theanaemic-domain-model-is-no-anti-pattern-its-a-solid-design/.
[8] "OWL 2 Web Ontology Language Mapping to RDF Graphs," OWL 2 Web Ontology Language Mapping to RDF Graphs. https://www.w3.org/tr/2009/rec-owl2-mapping-to-rdf-20091027/.
[9] G. Stevenson and S. Dobson, "Sapphire: Generating Java Runtime Artefacts from OWL Ontologies," Lecture Notes in Business Information Processing Advanced Information Systems Engineering Workshops, pp. 425–436, 2011.
[10] "Jena Semantic Web Framework", http://jena.apache.net/.
[11] "Object Model", http://objectmodel.js.org/
[12] "rdfstore-js", https://github.com/antoniogarrote/rdfstore-js