How To Simplify Building Semantic Web Applications

Matthias Quasthoff, Harald Sack, Christoph Meinel

Hasso Plattner Institute, University of Potsdam
{matthias.quasthoff, harald.sack, meinel}@hpi.uni-potsdam.de

Abstract. This paper formalizes several independent approaches on how to develop Semantic Web applications using object-oriented programming languages and Object-Triple Mapping. Using such mapping, Semantic Web applications have been developed up to three times faster compared to traditional Semantic Web software engineering. Results show that at the same time, developer satisfaction has been significantly higher if they used object triple mapping. We present a formal notation of object triple mapping and results of an experimental evaluation clearly showing the benefits of such mapping. The work presented here may one day help to make Semantic Web technologies part of the majority of future applications.

1 Introduction

Using and handling RDF data in software is not trivial to implement, especially with regards to the large number of best practices to consider. Before even that, developers need to learn about RDF concepts and how to deal with them using RDF programming libraries [1]. For many tasks, additional knowledge about RDF schema and OWL is required. This makes getting started with the Semantic Web quite a challenge for Semantic Web beginners and entry-level developers.

In general, two issues with current Semantic Web programming libraries can be identified: First, their complete potential is always visible to developers, instead of by default only revealing those parts that would be sufficient for a majority of implementation problems. Second, lots of RDF- and Linked Data-related implementation tasks, such as discovery, retrieval, and publishing of datasets need to be implemented by hand, resulting in huge implementation efforts even for relatively small implementation problem (cf. Section 4).

A promising approach for simplifying Semantic Web software development is Object Triple Mapping (OTM) [2, 3]. There do exist some OTM implementations, and also some research on its expressivity (cf. Section 2). However, there is no research on the actual building blocks needed to develop Semantic Web applications on top of existing RDF programming libraries, and there is no research on whether, or how OTM actually simplifies application development. With our work, we are confident to lay out guidelines for the development of future, easy-to-use Semantic Web programming libraries.

In Section 3, wie formalize OTM and extend it by a small pseudo-code vocabulary describing linked data functionality. This formalization is a starting point for further research in this important area. We used the formalization for an experimental evaluation of the benefits of OTM. The results of this evaluation are presented in Section 4. Section 5 concludes the paper.

2 Related work

Linked Data has become one of the most popular topics among the emerging Semantic Web [4]. Best practices need to be identified and described, how to efficiently implement linked data applications. Design patterns formalize such best practices; not as programming libraries, but as solutions to frequently recurring problems. They have been introduced to software engineering by [5]. In the world of relational database management systems, widely-used design patterns on object-relational mapping (ORM) have been identified [6]. These patterns allow developers to simply instantiate objects, and they are automatically filled with contents from a relational database. Modifications to these objects will automatically be persisted in the database.

The need of simplifying Semantic Web software engineering in a similar way to ORM has been identified in [2], and an implementation called So(m)mer, based on meta-programming, is available[1]. Up to now, several other tools for OTM have been inspired by object-relational mapping. The D2RQ Platform [7] uses a declarative language to describe mappings between relational database schemata and Semantic Web ontologies and lets clients access RDF views on the underlying non-RDF data. Other approaches such as RDFReactor[2] take mappings between OO programming units and Semantic Web schemas and allow software developers simplified access to a triple store. OntoJava [8] uses a similar approach to use auto-generated inference rules inside application source code. Winter [9] extends So(m)mer to allow for mappings of complex patterns instead of plain RDF classes onto Java classes.

All these solutions and the research done around them focus on specific parts of the approach chosen, such as feasibility and expressivity of such mapping, but do not yet aim at supporting the whole process of discovering, retrieving, processing, and re-publishing linked data, which will involve further steps such as, e.g., policy or data license checking [10]. To overcome this situation we have identified the principles common to these solutions and formulated design patterns for OTM, closely resembling ORM design patterns [3]. As our contribution in this paper, we will formalize the design pattern and evaluate how actual software can be built upon it.

---

[1] https://sommer.dev.java.net/
[2] http://semanticweb.org/wiki/RDFReactor

3 Formalizing OTM

It is good practice to encapsulate business or domain logic in classes and methods of object-oriented (OO) programming languages [6]. E.g., if a software product deals with people and relations between them, the software's object model likely contains a `Person` class and `friends` field for this class. To use RDF data in most OO programming languages, the mapping from RDF properties to the domain classes' fields has to be implemented by hand. Our hypothesis is that large parts of this OO handling of RDF concepts, including discovery and retrieval on the WWW, should be hidden from software engineers, making the development of Semantic Web software much easier, and hence encouraging software developers to actually start creating such software. In the following sections, we introduce a formal notation of the knowledge representation in OO programming, a mapping between RDF and OO concepts, and a simple pseudo-code vocabulary relevant for building applications using such mapping.

3.1 Basic concepts

The RDF data model has an established formal notation building upon the following concepts [1].

Definition 1 (RDF data model) *Let U be the set of* URI references, *B an infinite set of* blank nodes, *and L the set of* literals.

- *$V := U \cup B \cup L$ is the set of RDF* nodes,
- *$R := (U \cup B) \times U \times V$ is the set of all* triples *or* statements, *that is, arcs connecting two nodes being labelled with a URI,*
- *any $G \subseteq R$ is an* RDF graph.

How RDF graphs are actually constructed, handled, and transferred is subject to standards, conventions, and technical constraints. Linked data principles [4] suggest to provide smaller sub-graphs describing individual resources. In [11], an abstraction on top of these principles is described providing the whole Web of Data as one huge graph. Hence, operating on RDF data involves not only operating on triples and resources, but also retrieving the right sub-graphs of R, which will be described in a later section.

For OO programming, there is no single established formal notation focusing on the information representation part. Hence, we just use some basic formal concepts to capture OO environments form the information representation perspective.

Definition 2 (OO data model) *Let O be a set of* object identifiers, *F a set of* field names.

- *$S := \mathcal{P}(O)^F$ is the set of* field assignments $s : F \to \mathcal{P}(O)$,
- *$Q := S^O$ the set of* system states $q : O \to S$.

A system state $q \in Q$ maps each object $o \in O$ onto a field assignment $s :=$ $q(o) \in S$, which in turn maps each field name $f \in F$ onto the object's values $s(f) \subseteq O$ for this field. Note that in our notation, a field assignment returns sets of objects as field values. This allows to represent programming concepts such as array or collection objects. Ordered lists and formal cardinality and type restrictions (i.e., scalar-value fields or statically typed object definitions) on OO data models are outside the scope of this paper but can easily represented on top of our formalism if needed.

Example 1 (comparison of RDF and OO data model) *Let p_1, p_2, $p_3 \in U$ be URI denoting three people, $n \in U$ be the URI* `foaf:name` *and $k \in U$ be the URI* `foaf:knows`. *An RDF graph describing p_1 and p_2 might look the following.*

$$G := \Big\{ \langle p_1, n, \text{``John Doe''} \rangle, \langle p_1, k, p_2 \rangle, \langle p_1, k, p_3 \rangle, \langle p_2, n, \text{``Jane Doe''} \rangle \Big\}$$

Let us now have a look at this example from the OO perspective. Let o_1, o_2, $o_3 \in O$ be object identifiers denoting three people, `name`, `friends` $\in F$ *be field names, $q \in Q$ a system state, and $s_1 := q(o_1)$, $s_2 := q(o_2)$. The OO representation of G will look the following.*

$$s_1(\texttt{name}) = \{ \text{``John Doe''} \}$$
$$s_2(\texttt{name}) = \{ \text{``Jane Doe''} \}$$
$$s_1(\texttt{friends}) = \{ o_2, o_3 \}$$

3.2 Mapping RDF and OO

In this sections we continue to use the set definitions from the previous section.

Definition 3 (Object triple mapping, OTM) *An* object triple mapping *for an RDF Graph R, fields F and objects O is some (G, m_t, m_a, q) such that*

- *$G \subseteq R$ is an RDF graph*
- *$m_t : F' \to U$ for mapped fields $F' \subseteq F$ (the vocabulary map),*
- *$m_a : O' \to U$ for mapped objects $O' \subseteq O$ (the instance map)*
- *$q \in Q$ a system state such that for all $u \in U$, $o \in O'$, $f \in F'$ and $s := q(o)$*
  - *$|m_a^{-1}(u) \cap s(f)| \leq 1$*
  - *$|m_a^{-1}(u) \cap s(f)| = 1 \Leftrightarrow \langle m_a(o), m_t(f), u \rangle \in R$*

Note that this definition does not require the instance map m_a to be injective, which would be desirable in many cases, at least from a software engineer's point of view. However, there might be different simultaneous OO representations o_i of a single RDF resource u resulting from, e.g., different data licenses, trust policies, or access control decisions. Hence, the injectivity of the actual instance map presented to the developer should rather be ensured using additional formal representations of policies, contexts and the like, instead of being a general requirement to the instance map. Also, our notion of OTM does not

necessarily require a class map for RDF and OO, since many dynamically-typed object-oriented programming languages do not have the notion of classes. For statically-typed programming languages, an implementation of such class map will however be required. Treating other RDF concepts such as lists or reification, and also the semantics of RDFS and OWL are not part of this mapping, but subject to OTM implementations.

Example 2 (OTM) *To map RDF representations of people to the corresponding OO representation (cf. example 1), we need*

- *a vocabulary map m_t : `name` $\mapsto n$, `friends` $\mapsto k$;*
- *mapped objects o'_1, o'_2, $o'_3 \in O'$ such that $m_a : o'_1 \mapsto p_1$, $o'_2 \mapsto p_2$, $o'_3 \mapsto p_3$;*
- *a system state $q \in Q$ and field assignments $s'_1 := q(o'_1)$ and $s'_2 := q(o'_2)$.*

The mapped objects o'_1, o'_2 will exactly look like o_1, o_2 from example 1:

- *$s'_1(\texttt{name}) = \{$ "John Doe"$\}$ because $\langle \underbrace{m_a(o'_1)}_{=p_1}, \underbrace{m_t(\texttt{name})}_{=n}$ "John Doe"$\rangle \in G$*

- *$s'_2(\texttt{name}) = \{$ "Jane Doe"$\}$ because $\langle \underbrace{m_a(o'_2)}_{=p_2}, \underbrace{m_t(\texttt{name})}_{=n}$ "Jane Doe"$\rangle \in G$*

- *$s'_1(\texttt{friends}) = \{o'_2, o'_3\}$, because $\langle \underbrace{m_a(o'_1)}_{=p_1}, \underbrace{m_t(\texttt{friends})}_{=k}, \underbrace{m_a(o'_i)}_{=p_i} \rangle \in G$.*

Although several implementations of such mapping exist (cf. Section 2), for our research we use our own OTM implementation, which is available licensed under the GPL[3]. This implementation strictly follows the OTM design patterns derived from object-relational patterns [3, 6].

3.3 Building Linked Data software

Building a linked data application using OO programming will involve handling RDF resources as OO objects. There are only two ways to obtain objects from resources, and we introduce the following pseudo-code notation for them:

- `get`(u) for $u \in U$: Request $o \in O'$ such that $m_a(o) = u$;
- `query(pattern)`: Request $O'' \subseteq O'$ matching a `pattern`, using SPARQL.

From Definition 3, it is not clear how the RDF graph G is obtained. G will be constructed during application execution using the follwoing ways.

- Directly query a triple store, e. g., using SPARQL,
- `load`(u) for $u \in U$: Ensure that the dereferenced graph $G_u \subseteq G$,

Following linked data principles, an OTM implementation can automatically call `load`(u) on occurences of `get`(u). Just as this simplifies resource handling, two more pseudo-code operations are required to build linked data applications.

---

[3] http://projects.quasthoffs.de/otm-j

– `use(u)` for $u \in U$: Set up the OTM implementation to use the dataset u, i. e. evaluate the data license, and set up the SPARQL endpoint to be used. Even further decisions can be made, such as deciding upon some statistics whether to dereference single URIs for this dataset or rather to download a data dump.
– `publish(O'')` for $O'' \subseteq O$: Publish objects as linked data, either by producing serialized RDF files, or by hooking into some Web programming framework. By configuring meta-data for publication such as licenses, several checks, e. g. on license compatiblity, can be performed.

4 Evaluation

Our primary motivation for investigating OTM is to understand why Semantic Web technologies have been picked up so hesistatingly by software developers, and to show software developers how they can simply use and benefit from these technologies. We asked software engineers with little or no experience in Semantic Web software engineering (but yet sufficient programming skills) to solve a problem using RDF data sources and programming libraries.

4.1 Setup

Each participant was assigned two tasks, one of which was to be solved without OTM, and the other one using OTM. The order of the two tasks and the order of using/not using OTM was randomized to ensure the results will not be distorted by learning effects. Participants used the Eclipse programming environment and the jUnit framework to test their results[4]. To simulate the usual work-flow of Web programmers, we provided the participants with example source code of similar solutions [12].

Tasks. The experiment dataset consisted of 12,726 fictious *foaf:Person* resources, 100 *foaf:Document* resources, 15 *foaf:Group* resources, and 169 *bldg:Room* resources[5]. Each document had between 2 and 4 authors, each group between 8 and 15 members, and each person knew a number of other people. All resources could be dereferenced by their URI. The following tasks needed to be solved.

Task 1. Given a set of URI identifying documents, construct the set of all the documents' authors' names.
Task 2. Given a URI identifying a person, construct the set of all person's friends' friends' names.

The participants were expected to find a solution close to the following pseudo-code (using the vocabulary from Section 3.3), which was however not presented to the participants.

---

[4] http://eclipse.org/, http://junit.org/
[5] *foaf:* http://xmlns.com/foaf/0.1/, *bldg:* http://example.org/buildings/

Solution 1. `GET_AUTHOR_NAMES(publication_uris):`

```
load(dataset_uri)
for each uri in publication_uris
    publication = get(uri)
    for each person in publication.authors
        return person.name
```

Solution 2. `GET_SECOND_ORDER_FRIENDS(person_uri):`

```
load(person_uri)
person = get(person_uri)
for each friend in person.friends
    load(friend.uri)
    for each friend2 in friend.friends
        load(friend2.uri)
        return friend2.name
```

4.2 Metrics

Difficulty. After each of the two assignments, participants were asked to *estimate the difficulty* of the assigment, the *maintainability of the resulting source code*, and how difficult a solution would have been using *XML stores* or *RDBMS* instead of RDF, on a scale from 0 (trivial) to 10 (too hard). After the first assignment only (which randomly had to be solved either using OTM or without OTM), participants were asked whether they see potential use of RDF in their near-future projects. Along with these subjective measures, we tracked the *time* required to find a working solution, and the number of *edit-debug cycles*.

Source code. Since OTM encapsulates large parts of RDF data handling it is expected that solutions building on OTM will have less lines of code than solutions than non-OTM solutions. To get a deeper understanding of how lines of code will be reduced, we separately counted lines of code carrying

- *language constructs* such as loops, variable declarations etc.;
- *RDF library initialization code*, i.e. creation of or connection to data stores;
- *data access code*, i.e. imperative statements for URI handling such as `get`, `load`, `use`, or `query` operations, (cf. Section 3.3), and RDF concept manipulation using RDF libraries; and
- *business* or *domain logic*, i.e. lines of code manipulating domain objects such as people, publications or names, e.g., by handling or accessing fields of domain objects.

Feedback. Besides these rather quantitative metrics, we gathered feedback during and after the experiment. In a questionnaire, we asked the participants to identify the biggest problems during finding the solutions, and to name other types of support they wished they had.

4.3 Participants

Undergraduate computer science students at Hasso Plattner Institute, University of Potsdam were invited to to participate in the experiment. Ten participants aged 19 to 27 (mean 22.6) actually took part in the experiment. All participants had between 4 and 9 years of programming experience (mean 6.6). According to their estimation on a scale from 0 (none) to 10 (expert) prior to solving the assignments, only three of them said having some basic knowledge about RDF, the others none (mean 0.9). All participants were experienced with the Java programming language (mean 5.4). Also, most participants had some experience using traditional information stores such as XML documents (mean 3.4) and RDBMS (mean 3.9).

4.4 Results

Difficulty. The results for developer satisfaction were surprisingly clear (Fig. 1). Implementing the assignments was found to be significantly easier using OTM (mean 2.4, $\sigma = 1.8$) than using the Jena RDF library only (mean 6.4, $\sigma = 2.1$). Also, the participants judged their solution significantly easier to maintain (both for themselves: 1.13, $\sigma = 1.6$, and if they let somebody else do it: 2.13, $\sigma = 1.5$), if OTM had been used compared to non-OTM (means 4.3, $\sigma = 1.8$ and 5.9, $\sigma = 1.1$). However, whether the first assignment was to be solved using OTM or without OTM had no influence on the participants' estimation of the difficulty of integrating Semantic Web technologies in their own future projects (means 4 and 5). Regarding the estimated difficulties, we can eliminate variance by comparing the differences of the difficulty of the implementation and the estimated difficulties of alternative approaches using XML or RDBMS (Fig. 2). By means, the non-OTM solution has been rated more difficult compared to traditional data formats, whereas the OTM solution has been rated easier to implement than traditional data formats. However, the differences for these relative difficulties are not significant. For Task 1 (finding the names of publication authors), both the number of *edit cycles* (non-OTM mean 29, $\sigma = 6.6$) and the time needed to find the solution (mean 1.2 hours, $\sigma = 0.5$) was significantly lower using OTM (8.8 edit-debug cycles, $\sigma = 6.2$ and 0.4 hours, $\sigma = 0.2$, Fig. 3). For Task 2 (finding the names of friends' friends), the mean number of edit-debug cycles was increased from 8 to 19, while the mean time needed to find the solution was decreased from 0.6 hours ($\sigma = 0.3$) to 0.4 hours ($\sigma = 0.4$) using OTM. However, the differences for Task 2 are not significant. The combined figures for Task 1 and Task 2 show that the number of edit-debug cycles remains stable, but the development time has been decreased significantly using OTM. It is unclear why the number of edit-debug cycles is larger for Task 2 using OTM. But since the development time was not increased, we do not consider this a general weakness of OTM. It will however be interesting to direct further research in this direction.

Source code. The figures for the source code metrics are clear again (Fig. 4). The overall lines of code are reduced from 20.9 ($\sigma = 5.4$) not using OTM
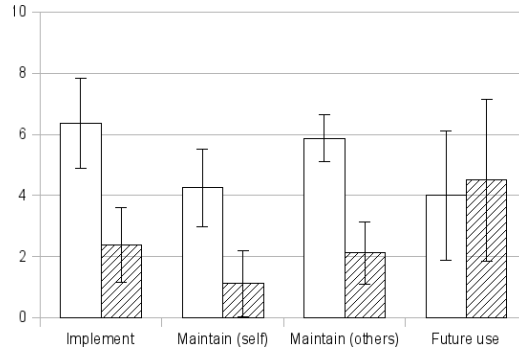
Fig. 1. Difficulty (0 – easy, 10 – too hard) of *implementing the solution*, and estimated difficulty of *maintaining the result's source code* and *using Semantic Web technologies in future projects* for non-OTM (white) and OTM (streaked). Error bars show 95% CI.
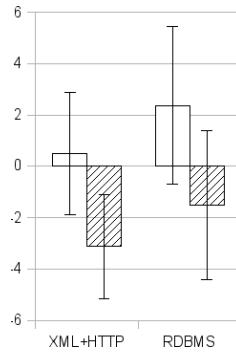

Fig. 2. The mean difference of the difficulty of *implementing the solution* and the and estimated difficulties of implementing alternative solutions using *XML and HTTP* or *RDBMS* is higher for non-OTM (white) solutions than for the OTM solutions (streaked). Error bars show 95% CI.

to 11.3 ($\sigma = 7.7$) using OTM. The number of lines of code carrying language constructs (mean 5.1 for non-OTM and 4.4 for OTM) and lines carrying business and domain logic (means 5.1 and 3.5) remain about the same, no matter if OTM is used or not. But lines of code for library initialization (non-OTM mean 4.8, $\sigma = 1.8$) and data access (12, $\sigma = 5.4$) are reduced significantly to 1.5 ($\sigma = 1.4$) and 2.8 ($\sigma = 3.9$) if OTM is used. This is plausible as using OTM no objects simply representing vocabulary (such as Jena's `Property`) or data access interfaces (such as Jena's `StmtIterator`) need to be instantiated. Additionally, using our OTM implementation the `load` operation can be omitted, as on calls to `get` and on field access `load` is called automatically. However, the main benefit of these implicit calls is not reduced lines of code, but improved separation of concerns, i. e. data access is separated from domain logic.
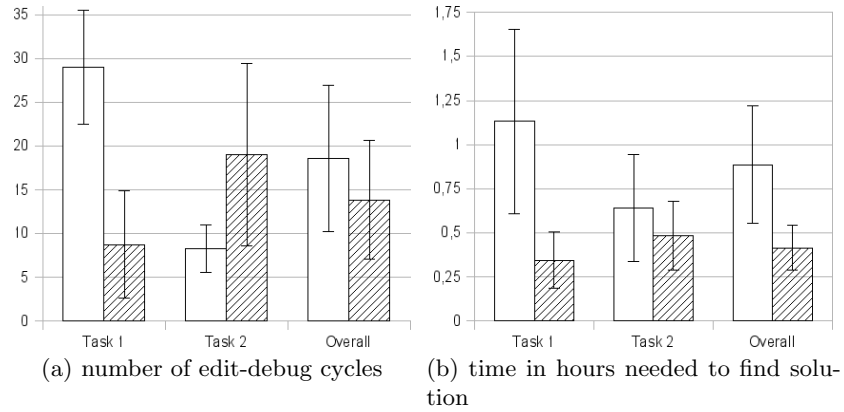
(a) number of edit-debug cycles   (b) time in hours needed to find solution

Fig. 3. Depending on the implementation task, the number of *edit-debug cycles* can be reduced significantly using OTM (streaked), compared to OTM (white), 3(a). OTM can also significantly reduce the implementation time of Task 1. In spite of increased number of edit-debug cycles for Task 2, the time needed to find the solution is not increased, 3(b). Error bars show 95% CI.
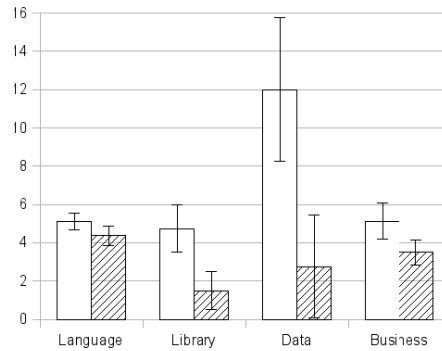

Fig. 4. The significant decrease in lines of code using OTM is achieved by reducing *library initialization* and *data access* code. Error bars show 95% CI.

Qualitative Feedback. Asked for the biggest problems they faced solving the non-OTM tasks, participants found it generally hard to understand the Jena API. Participants found it difficult to understand the central Jena classes `Model`, `Resource`, and `Property`, which is due to their lacking knowledge of RDF concepts. Also, participants had problems understanding `Model.read`, which loads RDF data from the URI specified, and `Model.getResource`, which only creates a `Resource` object to be further processed in Java. Some were unsure when a `String` in Jena was a literal value, and when it represented a URI. One participant commented "The source code contains weird objects, which do not have to do anything with the problem domain. This causes programming mistakes, because these objects are untyped." He mentioned the example of deciding whether `RDFNode` is a `Literal` or a `Resource`. Also, Jena's `StmtIterator`, which needs to be used in order to read triple information, was criticized for being confusing to use. All in all, participants highly valued the source code examples we provided, and said finding a solution would have taken much longer without the example. However, some would have preferred more comprehensive examples, featuring nested loops, or complete howto documents.

OTM received less comments, which is probably due to the fact that each participant was to solve two assignments–one using OTM, one without–and the OTM task was easier to solve than the non-OTM task. Still, we received valuable feedback. Participants found it hard to find the mapped Java class representing resources of a specific RDF type. Some participants have been observed analyzing all mapped Java classes available, others just read the example source code provided and concluded the right classes to use. However, one participant just guessed arbitrary (wrong) classes to be used in his source code and got stuck for a while. Although our OTM implementation simplifies URI dereferencing by implicitly loading RDF graphs when instantiating mapped objects, it took one participant a while to find out how to explicitly load a whole dataset as needed for Task 1. Some participants had trouble in dealing with generic Java types used for collections of objects, and hence could not fully benefit from our OTM framework.

Both the OTM and non-OTM solutions shared some comments. Participants said a graphical representation of the RDF schema, or the mapped class model, and a graphical browser for the dataset would have helped them to understand the data structures and would have improved their implementation performance. Also, participants complained about not having understood how or where the data had actually been stored. Only very few participants had the idea of viewing the URI provided by the test framework in a Web browser window.

5   Conclusion

In this paper, we presented a formalism for Object Triple Mapping (OTM), a promising approach to structuring the development of Semantic Web software. Our OTM formalism harmonizes several implementions seeking to simplify Semantic Web application development and adds process elements to describe

complete programs operating on linked data. Our second contribution is an experimental evaluation of OTM. We presented the results of this experiment, clearly showing that

- OTM speeds up the development of Semantic Web software.
- Lines of code needed to solve several tasks are reduced by half using OTM, and the share of "purely technical" lines of code is diminished so that using OTM, business logic and program structure stands out in the code.
- Improved programming experience can be measured, as developers without Semantic Web programming experience find it simpler to develop software using OTM, and are more satisfied with the quality of their results.

The experiment material, assignments, datasets etc. can be downloaded from the experiment web site[6]. We encourage readers to join the evaluation, and share their results with us. To obtain a broader view on what are the Semantic Web software engineers' pains, how we can help them, and which technology they actually prefer, we will extend our evaluation to more programming languages and RDF programming libraries. Besides this planned continuous evaluation, we will publish the direct and indirect feedback we receive from participants, and will incorporate that feedback into our own OTM implementation for further evaluation, and are willing to contribute to other existing OTM implementations.

As the results of our experiment are very promising, we are confident to contribute in further spreading the word about positive experience using Semantic Web standards and technologies. Once software engineers and managers are convinced that Semantic Web technologies can be introduced in software projects without adding costs, or even reducing costs, software will start to contain more and more Semantic Web technologies, fostering interoperability and data mash-ups. By then, software engineers will be willing to learn more about these technologies and more complex software projects going beyond the features of off-the-shelf OTM implementations can finally be done.[7]

References

1. Manola, F., Miller, E.: Rdf primer. w3c recommendation 10 february 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/ (2004)
2. Story, H.: Java annotations and the semantic web. http://blogs.sun.com/bblfish/entry/java_annotations_the_semantic_web (2005)
3. Quasthoff, M., Meinel, C.: Design patterns for the web of data. In: Proc. of IEEE SCC 2009. (2009)
4. Berners-Lee, T.: Linked data. http://www.w3.org/DesignIssues/LinkedData.html (2006)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)

---

[6] http://hpi-web.de/meinel/quasthoff/otm-experiment-2009-06

[7] For the cited work still to appear ([13] and [3]), please find the electronic version at http://hpi-web.de/meinel/quasthoff.

6. Fowler, M., Rice, D.: Patterns of Enterprise Application Architecture. Addison-Wesley (2003)
7. Bizer, C., Seaborne, A.: D2rq – treating non-rdf databases as virtual rdf graphs. In: Proc. of the 3rd International Semantic Web Conference, Springer (2004)
8. Eberhart, A.: Automatic generation of java/sql based inference engines from rdf schema and ruleml. In: Proc. of the 2nd International Semantic Web Conference, Springer (2002)
9. Saathoff, C., Scheglmann, S., Schenk, S.: Winter: Mapping rdf to pojos revisited. In: Proccedings of the ESWC 2009 Demo and Poster Session. (2009)
10. Miller, P., Styles, R., Heath, T.: Open data commons, a licence for open data. In: Proceedings of the WWW2008 Workshop on Linked Data on the Web, Springer (2008)
11. Hartig, O., Bizer, C., Freytag, J.C.: Executing sparql queries over the web of linked data. In: Proc. of the 8th International Semantic Web Conference, Springer (2009)
12. Brandt, J., Guo, P.J., Lewenstein, J., Klemmer, S.R.: Opportunistic programming: How rapid ideation and prototyping occur in practice. In: Proc. of the Fourth Workshop on End-User Software Engineering, ACM (2008)
13. Quasthoff, M., Sack, H., Meinel, C.: Can software developers use linked data vocabulary? In: Proc. of I-Semantics '09. (2009)