Undefined 0 (0) 1
IOS Press

Reengineering application architectures to expose data and logic for the web of data

Juan Manuel Dodero [a,*], Ivan Ruiz-Rube [a] and Manuel Palomo-Duarte [a]
[a] *School of Engineering, University of Cadiz, Av. de la Universidad 10, 11519 Puerto Real, Cadiz, Spain*
E-mail: {juanma.dodero,ivan.ruiz,manuel.palomo}@uca.es

Abstract

This paper presents a novel approach to reengineer legacy web applications into Linked Data applications, based on the knowledge of the architecture and source code of the applications. Existing application architectures can be provided with linked data extensions that work either at the model, view, and controller layer. Whereas black-box scraping and wrapping techniques are commonly used to add semantics to existing web applications without changing their source code, this paper presents a reverse engineering approach, which enables the controlled disclosure of the internal functions and data model as Linked Data. The approach has been implemented for different web development frameworks. The reengineering tool is compared with existing linked data engineering solutions in terms of software reliability, maintainability and complexity.

Keywords: Software Architecture, Linked Open Data, Model-View-Controller

1. Introduction

The Web of data is largely concerned with procuring web applications that publicly display their information by means of metadata and explicit semantics, such that entities from heterogeneous web information systems can be interlinked [17]. Best practices of Linked Open Data (LOD) software engineering set out how data from web sources must be published and connected [18]. There is a challenge for legacy web applications that do not expose their data using the common formats and protocols of the Web of data, in order that their resources can be easily discovered and linked.

The Linked Data (LD) principles provide a guide to *reengineer* legacy web *apps*, thus preparing these for the Web of data. According to these principles, providing an existing web application with LOD capabilities encompasses a number of steps, including: to define a public LD metadata schema; to compile and link the application data instances; and to provide an Application Programming Interface (API) that enable third parties to browse the application resources based on the public metadata and semantics [15]. Such steps are usually engineered as external software components (e.g. web scrappers, data wrappers and diverse middleware) as an extension to the legacy application architecture. Lots of LD techniques, as well as software tools and frameworks supporting the creation and linking of RDF metadata from existing data sources, web sites and applications have been provided. Such techniques range from mapping relational data sources [34] to interlinking datasets [37] to exposing linked data APIs [14], among others.

Without a methodology and tool support, the migration from existing web sources and applications to LD-enabled ones can be a costly process, having a not insignificant risk [36]. There is a need to provide strategies to reengineer such legacy systems while preserving diverse software quality features, particularly concerning with non-functional aspects (e.g. security, privacy, reliability, maintainability, complexity, etc.) of the reengineered system. This paper presents a novel approach to reengineer legacy web applications into LD applications with this aim. Whereas typical scraping and wrapping techniques are commonly used to add semantics to existing web apps without changing

[*]Corresponding author. E-mail: juanma.dodero@uca.es.

their source code, a more intrusive approach that also exposes the apps' internal structure and data model as LD is presented here. Reverse engineering requires knowledge of the application source code. As opposed to the black-box approach of adding LD or converting an applications' output to LD without modifying the internals of the app, the white-box reengineering approach modifies the application itself by generating the software components that are required to implement the LD extensions. Such extensions can operate either at the data or the API level of the web application.

The research methodology followed for this aim is based on the design and creation strategy, as defined by Oates [25]. This strategy is suitable for research in Computing when the result is a tangible product. The design research process involves five steps, namely: awareness, suggestion, development, evaluation and conclusion [35]. In Section 2, the existing LD reengineering approaches are classified, articulated and analyzed as part of the awareness step. Then we conclude with the suggestion of the new approach, based on the discernible software architecture of most web applications. In Section 3, the new approach to develop LOD extensions is explained. The implementation and its evaluation is explained in Section 4, followed by a consolidated discussion of the findings and limitations of the research in Section 5. Finally, Section 6 presents some conclusions and future lines of work.

2. Reengineering Linked Data Application Architectures

The architecture of LD applications are discussed by Heath and Bizer [16] as a means to structure the components that the LD software system comprises. The most widespread Linked Data Application Architecture (LDAA) is the crawling pattern, which is suitable for implementing LD applications over a growing set of resources that are crawled and discovered. Alternative patterns have a very low query execution (e.g. browser-based dereferencing pattern) or they are used only when the number of data sources is small (e.g. the query federation pattern). On the top layer, the crawling pattern is made up of a *data access, integration and storage layer*, made up of a set of pipelined modules (i.e. web access, vocabulary mapping, identity resolution and quality evaluation), over which an RDF API or SPARQL endpoint is provided. An extension to this architecture has been implemented based on a LD API layer on top of the data access and integration layer, which mediates between consumer applications and an integrated database [14]. Figure 1 depicts the original LDAA with the modifications added by Groth et al [14].

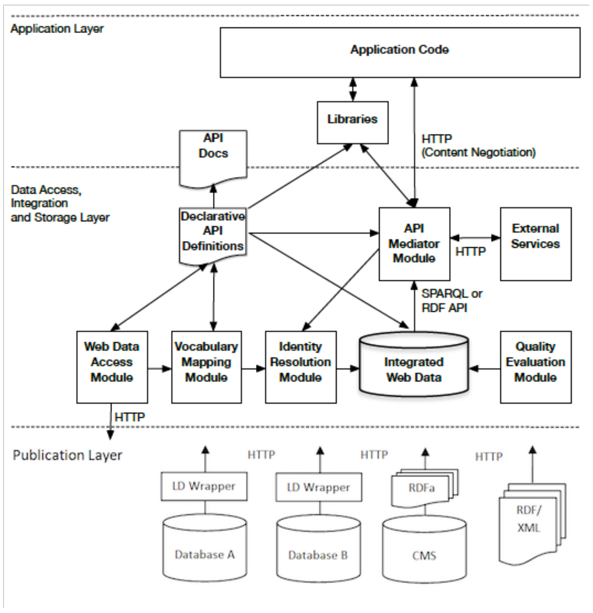

Figure 1. Extension to the original linked data application architecture by Heath and Bizer [16], with the additional functions to the data, integration and storage layer concerning API definition and mediation [14].

Eventually, pipelining all the functional modules of the data access and integration layer leads to the integrated database, which feeds the SPARQL endpoint or API mediator module. In the bottom, the *publication layer* usually implements wrapper modules that, either by scraping [28] or enriching [21] web resources, add the required semantics to existing resources and datasets. Setting up a middleware module is a common strategy to reengineer existing sources, which can range from HTML pages to structured data to web APIs [26]. Other scraping approaches harvest semi-structured HTML content and automatically convert it into structured LD instances [23]. Scraping and data wrapping techniques either convert data to LD or provide an API to access data [18]. The function of all such implementations can be provided either at the model, view, or controller level of an MVC architecture, as explained next.

– Apache Stanbol, KIM and SDArch [21] are examples of semantic enriching procedures that operate at the view level. OWL-S [29] and SA-REST

[33] can be used to describe web service properties at this level.

- The D2Rq server [4], Triplify [3] and Virtuoso RDF Views [9] are useful approaches to build wrappers at the data binding level. ActiveRDF can be used to align an application ORM with a given RDF schema [27].
- Middleware implementations, such as Virtuoso sponger [10] and Pubby, work at the controller level. Hydra [22] is a middleware implementation that also provides clients with JSON-LD descriptions of a new vocabulary, able to express common concepts of Web APIs. Other solutions, such as the Datalift platform [32] rely upon existing tools such as Silk [19] to provide interlinked RDF datasets.

In order to articulate the research issue, some of the linked data tools and frameworks enumerated above have been used by the authors to implement diverse LD information systems for a number of disciplines, such as information science [20] and software process management [31]. As a consequence of such development efforts, some practical engineering considerations for developing LD-enabled applications have emerged. These methodological aspects are summarized as the reengineering strategies described below. This study will be used thereafter to analyze the most relevant works related with LDAA reengineering, with a special focus on the Model-View-Controller (MVC) architecture [11].

2.1. *Reengineering strategies*

The LD reengineering strategies can be classified according to a number of features of the application, namely the availability of source code, the supply of a built-in information exposure facility (i.e. an external API), and the disclosure of database contents. The reengineering effort can range from a seamless integration of apps that already provide an LD API, to more costly adaptations for others that might not use any machine-friendly data format or standard protocol.

- *LOD scraping* [21,28]: This strategy applies if the web app source code and its internal data storage are not available at all, probably because the application was not initially designed to be reused by third parties. Information retrieval and web scraping techniques are the only alternatives to enrich their resources.
- *Data LOD wrapping* [3,4,9,32]: sometimes the web app source code is not available, but the application data is stored and available in a standard database format. Then, adapters or data wrappers can transform LOD requests into queries to the application data storage. Depending on the kind of storage, queries can be issued to database systems, structured files or any other data storage used by the application.
- *API LOD wrapping* [1,2,8]: sometimes the web app already provides an external API to reuse its data. A proxy, wrapper or middleware is implemented, so that LOD requests are formatted for the API and issued forth and back. The wrapper or middleware can add some data transformations and adaptations to the original API operations.
- *API LOD extension*: if the web app does not provide an external API, but its source code is available, a software add-on can be implemented to provide a LOD API. In this case, data and business models can be discovered from source code analysis of the MVC implementation. On one hand, if applied at the *model* layer, the extension strategy should generate a LOD schema from the internal data model implementation. The schema and data instances can thus be revealed through an external API.[1]. On the other hand, if applied at the *controller* level, the API extension strategy can use the existing implementation in order to avoid code duplication. Extending the API does not consist only in wrapping the existing controller implementation (i.e. the internal API) to make it public as a functionally equivalent external API. The requirements of the external API may not coincide with the internal one[2]. With the API extension strategy, extended functions can be implemented as an enriched API, likely with different non-functional features[3].

[1]The local namespace for the schema generated in this way initially reflects the app internal data model. Yet it can be aligned with standard LOD vocabularies through user-defined configurations, just like data LOD wrapping do, before making it public in the API.

[2]For instance, a legacy application internal API may implement a *finder* method that returns all objects of a given type. The external API, however, may require to define an additional *finderBy* method that returns only the objects that fulfill a given filtering condition.

[3]For example, different access privileges can be granted for the *finder* and *finderBy* methods of the external API

2.2. Reengineering MVC applications

The MVC architectural pattern is the most frequent for server-side web applications [6]. This section deals with how the LOD extension strategies can make a contribution to reengineer LOD-enabled versions of MVC-based legacy applications.

Figure 2 summarizes four reengineering strategies when applied to the MVC architecture. The figure portrays the practicable points of reengineering in the components of an MVC-based application. Leaving aside web scrapping, the rest of strategies can be applied at the three layers of MVC architecture, as explained next.

1. At the web view layer, metadata snippets (e.g. RDFa or microdata) can be added to the HTML content generated by the view components. Such snippets can be obtained by *scraping* web contents of the rendered views. Web view enrichment can be also applied for web services to describe web service properties at the view level [29,33].
2. At the data model layer, linked data can be generated from the underlying database binding by mapping all the data instances and the data schema. There are many approaches to build wrappers at this level. A slightly different strategy of data wrapping is to inspect the configuration of the Object-Relational Mapping (ORM) components, which are usually based on the ActiveRecord or the Data-Access-Object patterns [11].
3. At the controller layer, linked data can be provided as long as the controller operations can be adapted or extended to deliver equivalent LOD operations. These are normally designed with a ReST philosophy in mind [2]. Extending the ReST controller operations implies building a new API wrapper, which can be implemented either as a plug-in within the bounds of the web app, or as a separate middleware. Middleware implementations are thus aware of the web API instead of the inner app details. Regular applications are not often prepared with an architecture based on plug-ins or add-ons. The Hydra web API [22] allows to explicitly program API extensions and expose a precise LOD information model that can be exploited by an external browser.

3. The EasyData LOD extension strategy

The LOD extension approach is based on reverse engineering the source code of a web app in order to automate the LOD metadata provision and prepare it with a machine-friendly access API. *EasyData* is the name of a new approach to LOD extension in order to reengineer legacy MVC-based web apps. The reengineering cycle consists of a number of steps to implement diverse functions, which are mapped to the modules of a regular LDAA [14] as explained next:

1. *Revealing* the underlying application data model: An RDF model equivalent to the application ORM schema is generated and published as metadata. In addition, when the application receives a request, RDFa and microdata annotations are generated and embedded into HTML views. To facilitate external linking with standard vocabularies, a set of mapping options can be configured. In this stage, the functionalities of the *web access* module and *vocabulary mapping* module of the LDAA are developed.
2. *Linking* the application data instances: The linked datasets retrieved from the internal data storage of the application can be explored and linked. Internal data items can be readily linked. Afterward, an external interlinking module can be used to link external resources [37]. A study on how interlinking tools can help data publishers to connect their resources to the Web of data can be found elsewhere [30]. In combination with an interlinking tool, this phase develops the functionalities of the *identity resolution* module of the LDAA.
3. *Publishing* the service API: A service-oriented controller API that follows the Create-Read-Update-Delete (CRUD) pattern to consume the LD resources is generated in this phase. Hence extended service descriptions can be produced, based on OWL-S, SA-REST, Hydra or any equivalent scheme used to specify web service semantics. Yet the legacy application might have controller operation implementations that are useful for the service API. If it is allowed and secure to make such implementations public, they can be revealed in this phase, which thus implements the *application layer* of the LDAA.
4. *Controlling* the non-functional quality aspects. The non-functional requirements of the application that are part of the *quality evaluation* module

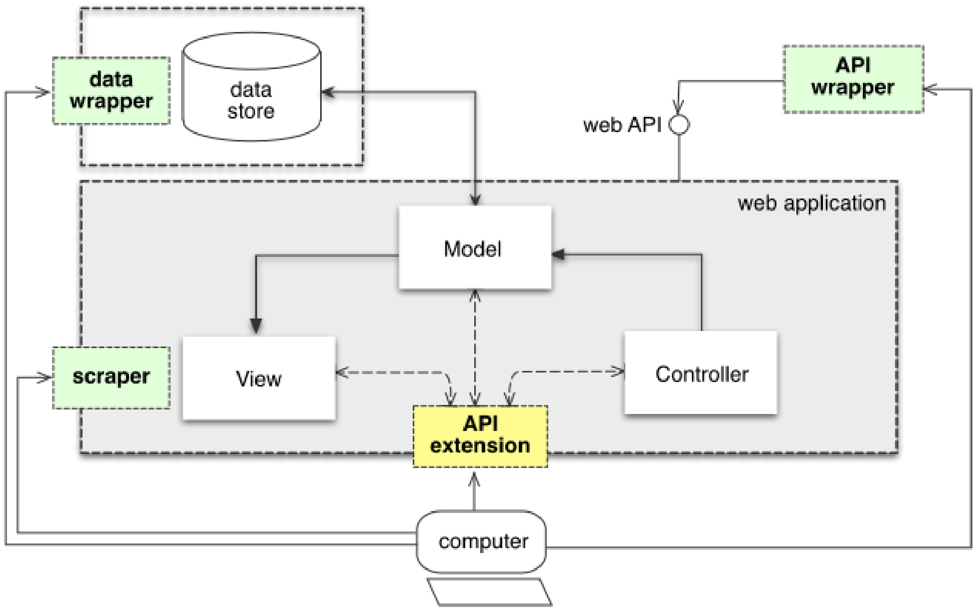

Figure 2. Applicable reengineering points in the MVC architecture to adapt a legacy web app

of the LDAA can be considered in this phase. For instance, security and access control to the information resources is a recurrent non-functional aspect for any software system. Since access control to the generated LOD resources should not be granted for everyone, authorization permissions can be defined for data items, data types and service operations in this step. Other non-functional aspects, such as data privacy, can be also considered here. For the sake of simplicity, therewith the security access control issues will be used to illustrate such aspects.

Web applications can be reengineered at the three layers of the MVC architecture, namely the data binding model, the web view and the business logic controller. The EasyData procedure is practicable as long as the application source code is available. This is granted in open source software but, in general, it requires an agreement with the software license owner. Actually, some type of permission might be required also for scrapping web contents, but this is out of scope of the LOD extension strategy.

4. Evaluating the LOD extension approach

Two different prototypes have been developed as evaluation case studies of the LOD extension strategy. Each prototype serves the EasyData procedure to be applied to different web development languages and MVC-based open source frameworks, such as *Ruby-on-Rails* and *Django*, which underpin the architecture of a considerable number of MVC-based web apps.

- The first implementation is called EasyData/Rails[4], which is a Ruby component or *gem* that, when installed in a ruby-on-rails app, reengineers the source code to create a LOD extension that wraps view, data and controller logic.
- The second implementation is called EasyData/Django[5], which is a Python add-on that generates similar LOD extensions for Django-based applications.

How the revealing, linking, publishing and controlling steps are performed with EasyData is explained next. The Redmine project management tool is used as an example of legacy web application to test the EasyData approach, which is made up of the following steps:

Step 1 – Revealing the application data model. The first step is to generate and publish an RDF model equivalent to the application data model. A simplified data model of Redmine is formed by the *Project*,

[4]https://github.com/dodero/EasyData_Rails
[5]https://github.com/dodero/EasyData_Django

Issue, *User* and *TimeEntry* classes, as illustrated in the Rails ORM implementation of Figure 3. EasyData can render the RDF model from the web app source code, as shown in Figure 3. The configuration option `set_rdf_model_name` defines the alignment of application data model elements with the concepts and properties of a standard RDF schema. In this example, the Redmine `Project` objects are mapped to DOAP *project*s, the Redmine `User` objects are mapped to FOAF *person*s, and the Redmine `TimeEntry` objects are mapped to OWL-Time *duration*s. Redmine `Issue` objects are not mapped to elements from an external vocabulary, since no standard model has been found that defines appropriately what a tracking issue is.

```
Namespace.register(
   :doap, "http://usefulinc.com/ns/doap#")
Namespace.register(
   :foaf, "http://xmlns.com/foaf/spec")
Namespace.register(
   :time, "http://www.w3.org/TR/owl-time")
class Project < ActiveRecord::Base
   has_many :issues
   set_rdf_model_name "doap:Project"
end
class Issue < ActiveRecord::Base
  @status = IssueStatus::OPEN
  belongs_to :project
end
class TimeEntry < ActiveRecord::Base
  @spent  = 0
  set_rdf_model_name "time:DurationDescription"
end
class ProjectTimeEntry < TimeEntry; end
class IssueTimeEntry < TimeEntry; end
class User < ActiveRecord::Base
  has_many :projects,
           :through => :projectTimeEntries
  has_many :issues,
           :through => :issueTimeEntries
  set_rdf_model_name "foaf:Person"
end
```

Figure 3. Revealing the application data model with EasyData/Rails

After revealing and mapping the data model, a customized set of template tags provided by EasyData can be embedded on the HTML web contents to generate RDFa or microdata snippets that enrich the web view. First, the data model must be loaded using the `load easydata_rdfa` or `load easydata_microdata` template tags. Then, a number of `instance` tags with the `rdfa_` prefix can be used to generate RDFa snippets of a given data instance. For example, `rdfa_div instance` and `rdfa_div_span instance` template tags generate RDFa-enriched HTML `div` and `span` elements; `rdfa_ul` and `rdfa_li` template tags generate RDFa-enriched HTML unordered lists. Likewise, template tags with `microdata_` prefix generate microdata-enriched versions of the same elements. Two sets of template tags are available, one with the `rdfa_` prefix and another equivalent with the `microdata_` prefix. Additional template sets can be easily implemented to generate JSON-LD or other formats without changing the basics of the approach.

EasyData template tags render by default all the visible properties of the data element that were previously revealed and mapped. If an HTML page needs to be enriched with the value of a particular property, the `_field` suffix followed by the property name can be appended to the `instance` template tag.

Step 2 – Linking application data instances. The datasets retrieved from the web application database can be configured to be directly linked as they are rendered in the HTML view. For example, EasyData can generate an HTML anchor code with an RDF instance URI by means of the `easydata_include_link instance` tag.

External linking targets can be added to the engineered application by means of such template tags. Instead of linking to the inner application model entities revealed in the previous step, the application view can be provided with links to other entities discovered by an external interlinking tool. For example, Table 1 shows the output of an interlinking process configured to match the Redmine data revealed by EasyData with a DBPedia dataset. EasyData template tags can be used to include links to these DBPedia entities in the application view.

Table 1

Output of an interlinking tool that matches a web app data model with dbpedia-en. Namespaces have been omitted for clarity

Resource ID	**title**	**DBPedia label**
resource#project	Project	resource/Project
resource#issue	Issue	resource/Issue_tracking_system
resource#user	User	resource/User_(computing)

Step 3 – Publishing the service API. The LOD extension can generate semantic descriptions of operations for a ReST-based service API. Figure 4 shows the enriched SA-REST description of the `getTimeSpentOnIssue` service operation, which returns a list of Redmine `TimeEntry` objects for each time slot that a `User` dedicated to a task or `Issue`. The `TimeEntry`, `User` and `Issue` elements are defined in an OWL model delivered as part of the earlier revealing and mapping stage.

```
<p about="http://my.example.org/redmine/getTimeSpentOnProject/">
The logical input of this service is an
<span property="sarest:input">
   http://my.example.org/ontologies/Redmine.owl#User
</span>
object and an
<span property="sarest:input">
   http://my.example.org/ontologies/Redmine.owl#Project
</span>
object. The logical output of this service is a list of
<span property="sarest:output">
   http://my.example.org/ontologies/Redmine.owl#TimeEntry
</span>
objects. This service should be invoked using an
<span property="sarest:action"> HTTP GET </span>
...
<meta property="sarest:operation"
  content="http://my.example.org/ontologies/Redmine.owl#timeSpentOnProject"/>
</p>

<p about="http://my.example.org/redmine/getTimeSpentOnIssue/">
The logical input of this service is an
<span property="sarest:input">
   http://my.example.org/ontologies/Redmine.owl#User
</span>
object and an
<span property="sarest:input">
   http://my.example.org/ontologies/Redmine.owl#Issue
</span>
object. The logical output of this service is a list of
<span property="sarest:output">
   http://my.example.org/ontologies/Redmine.owl#TimeEntry
</span>
objects. This service should be invoked using an
<span property="sarest:action"> HTTP GET </span>
...
<meta property="sarest:operation"
  content="http://my.example.org/ontologies/Redmine.owl#timeSpentOnIssue"/>
</p>
```

Figure 4. Publishing the service API. Namespaces are not included for clarity

Likewise the SA-REST description, other web service metadata based on OWL-S or Hydra Web API can be generated.

Step 4 – Controlling security access. Access control grant can be configured for data items, data types and service operations generated the previous steps. Figure 5 is an example of how the MVC controllers of EasyData are configured with the Rails `has_permission_on` and `filter_access_to` primitives. It permits access to `Project` and `Issue` resources as well as the `getIssues` and `getAssignedIssues` operations in a Redmine installation. The example defines access permissions for specific user roles (e.g. admin, analyst, etc.) and specific operations (e.g. create, read, update, delete). This configuration of the non-functional security features is part of the Rails implementation and does not need to be repeated elsewhere in external LOD wrappers, thus reducing *dispensable* code smells.

4.1. Comparison with other LDAA reengineering approaches

Reengineering an LDAA from an existing web app requires using an LD framework or toolset like the ones discussed above. In this section we have compared well-known LD frameworks and tools with EasyData on a number of software metrics provided by the SonarQube[6] code analysis platform. The tools have been selected as long as they fulfill a number of conditions, namely: the open source code is available, the implementation language can be analyzed in SonarQube, and they have component modules that implement either data or API wrapping.

The software metrics used to assess the different solutions enable to compare software reliability and security, maintainability, and size and complexity features for each framework. Except for the size metrics,

[6]`https://www.sonarqube.org/`

```
# controllers
class ProjectController < ApplicationController
   filter_access_to :all
   filter_access_to :getIssues,
                    :require => :read
   def getIssues ... ;  end
end
class UserController < ApplicationController
   filter_access_to :all
   filter_access_to :getAssignedIssues,
                    :require => :read
   def getAssignedIssues ... ;  end
end

# authorization_rules.rb
authorization do
   role :admin do
      has_permission_on :projects,
        :to => [:create, :read, :update, :delete]
      has_permission_on :issues,
        :to => [:create, :read, :update, :delete]
      ...
   end
   role :analyst do
      has_permission_on :projects,
        :to => [:read, :getIssues]
      has_permission_on :issues,
        :to => [:read, :getAssignedIssues]
   end
end
```

Figure 5. Controlling security access with EasyData configurations in ruby

the reliability, maintainability and complexity metrics provided by SonarQube are language-independent, so the tools can be compared despite their implementation language:

- *Bugs*: the number of bugs as a measure of software reliability.
- *Vulner*: the number of known vulnerabilities found as a measure of software security.
- *Code smells*: the number of code smells, i.e. a symptom in the source code that possibly indicates a deeper problem.
- *Technical Debt (TD)*: effort to fix all maintainability issues, measured as *hours* of required work to remediate the issues, or the *ratio* between the cost to develop the software and the cost to fix it. The ratio is computed as the remediation cost divided by the development cost. The development cost is estimated as 0.06 days (i.e. nearly 30 minutes) per line of code, by following the Mythical Man Month principles [5].
- *Code duplication*: amount of code *blocks* and code *lines* involved in duplication. The *ratio* is a measurement of the density of duplication, i.e. the number of duplicated lines divided by the overall number of lines.
- *LOC*: Lines of Code as a simple measure of the program size.
- *statements*: number of statements in the target language; SonarQube unifies this metric and makes it independent of the parsed language.
- *code duplication density*: density of duplicated lines, i.e. number of duplicated lines / LOC.
- *Complexity*: The Cyclomatic Complexity (CC) [24] is calculated based on the number of control flow paths through the code. SonarQube varies slightly the calculation, depending on the implementation language.
- *CC Density (CCD)*: CCD is measured as the average CC per statement in the source code; it provides a program size-independent measurement of complexity, which is demonstrated to be a useful predictor of software maintenance productivity [13].

These metrics are not completely independent from each other. For example, some size and complexity metrics are a clear indicator of software maintainability [13]; code duplication is also a kind of code smell known as *dispensable* (i.e. a portion of unneeded code whose absence would make the code cleaner, more efficient and/or easier to understand), which is also related with the technical debt. The reason to disclose such metrics separately is to check the reliability and maintainability of the different software solutions due to different causes that might be subject of improvement.

Particularly, the more complex software frameworks implement more functions than smaller tools. Consequently, the complete source code must not be analyzed for every framework. The source code analysis of each tool or framework only concerns the software modules that are concerned with the wrapping and LD conversion functions. Some tools are small and only perform such functions, but the source code of bigger frameworks had been inspected in detail to filter out modules that implement non-comparable things. The excluded modules were mainly the ones implementing the functions of the data access, integration and storage layer of the LDAA (e.g. Sesame SPARQL implementations and Silk interlinking library use, among others) as well as tool-specific functions unrelated with the rest of tools (e.g. Stanbol's semantic enrichment of contents). The list of modules that have been included in the analysis of each tool can be examined in the appendix A.

Table 2

Maintainability and code duplication of EasyData compared to other LD reengineering approaches

Tool	**Lang**	**Maintainability & Code Duplication**					
			TD		*Code duplication*		
		#code smells	*hours*	*ratio*	*#lines*	*#blocks*	*density*
D2Rq	java	805	84.5	1.25%	555	31	3.93%
Stanbol	java	0	44.9	1.99%	229	16	4.87%
HydraBundle	PHP	107	15,2	1.35%	369	4	15.68%
Triplify	PHP	166	18.5	2.85%	0	0	0
Datalift	java	1028	112.3	1.46%	2382	53	14.85%
EasyData	python	21	9.1	0.50%	687	79	18.21%

Table 3

Size and complexity of EasyData compared to other LD reengineering approaches

Tool	**Lang**	**Size & Complexity**				
		Size			*Complexity*	
		LOC	*#statements*	*#functions*	*CC*	*CCD*
D2Rq	java	14108	6473	1516	3239	0.50
Stanbol	java	4701	1887	352	724	0.38
HydraBundle	PHP	2354	1098	187	476	0.43
Triplify	PHP	1352	818	79	398	0.49
Datalift	java	16037	7009	1349	3043	0.43
EasyData	python	3773	2195	133	478	0.22

Table 4

Reliability and security of EasyData compared to other LD reengineering approaches

Tool	**Lang**	**Reliability & Security**				
				Remediation (h)		*Remediation*
		#bugs	*#vulner*	*bugs*	*vulner*	*effort (h)*
D2Rq	java	19	205	17	240	445
Stanbol	java	28	415	22	260	675
HydraBundle	PHP	14	280	0	0	280
Triplify	PHP	5	100	1	30	130
Datalift	java	51	430	96	1175	1605
EasyData	python	39	335	0	0	335

As shown in Tables 2 and 3, the EasyData implementation considerably reduces CC and TD values. Since such measurements are dependent on the program size, it is more accurate to observe the TD ratio and CC density to compare different solutions. In this vein, the TD ratio and CCD are lower for EasyData than for other solutions. The reduced TD has an influence in the maintainability of the solution.

On the other hand, all solutions present a smaller code duplication density than EasyData (see code duplication metrics in Table 2. Some of them (i.e. Hydra and Triplify) also present a better reliability and security remediation cost, measured as remediation hours required to fix bugs and vulnerabilities. That means a need for improvement of the EasyData implementation. Yet the number of vulnerabilities (see Table 4) is fewer for EasyData and HydraBundle, mainly because they make a less intensive use of existing libraries and components that might add security issues.

With respect to the size and complexity of each solution (see Table 3), HydraBundle, EasyData and Triplify have less complex implementations than other frameworks. Tools like D2Rq and Datalift add an extra complexity, because they make use of a lot of handful java libraries that have to be properly integrated and managed in the source code. This criteria is less relevant for EasyData, since the source code of the eventually extended LOD application is automatically generated, so the need to manage code complexity issues, which have to do with the programmers' difficulty for code maintainance, is less relevant in this case.

5. Discussion

This paper deals with how LOD extensions can be built for legacy web applications, while complying with a number of non-functional quality features that might be defined. The aim is not to define a new technique for linking heterogeneous LD schemata and LOD datasets, which could be made by means of existing techniques and tools, as explained elsewhere [37]. Instead, the goal is to describe a new approach to disclose data and resources from legacy web applications in a controlled way, which takes into account the server-side architecture of the application.

Some approaches, as the TAO method and tools [36], have been used to migrate legacy applications to semantic versions by deriving an ontology from the documents (e.g. specifications, diagrams, software manuals) that describe the application architecture. The transition of such legacy systems to semantic versions has missed a first-class element of software architectures, namely the source code. Considering source code is relevant to obtain semantic and LD versions of a legacy web application, especially because of the great number of web apps that have been recurrently designed and implemented on the basis of the MVC software architecture. The hypothesis of the present paper is that the reverse engineering of such well-known application architectures can be an advantage to provide reliable and maintainable LOD extensions to legacy web applications and systems.

Regular wrapping strategies pose a challenge to the design of web applications when non-functional requirements, such as security and access control, have to be changed or extended. These requirements are part of the business logic of the application. The issue is that web apps are not usually designed to implement their business logic in the data storage (e.g. as database stored procedures), but in intermediate controller components instead. Since third party applications do not necessarily have shared access to the controller components, providing extended LOD versions at the data binding layer can pose a design-level impediment. Therefore external browsers might have not the appropriate authorization privileges to the linked data emanating from the application data layer. Besides, duplicating the controller logic in the extended LOD version to implement changeable security access privileges does not reuse the application source code.

The EasyData reengineering method combines the Data LOD wrapping and API LOD extension in an integrated approach, which exposes the application data model and generates a LOD API, providing also a hook to implement diverse non-functional requirements. In its current implementation, EasyData provides an unified security access control layer, similar to the available in other frameworks. Thus external agents can be properly authorized to browsing LOD and accessing the application resources and API.

A major concern of the EasyData white-box approach is that the described LOD extension strategy is deeply integrated with the internal data model and logic of the legacy web applications. This tight coupling eliminates the separation of concerns in a separate LD layer and implies that changes to the inner workings of the web app may affect the EasyData plugin implementation. In practical terms, this can make maintenance more difficult if the original web app source code is not under the control of the LOD extension developer. For instance, in the Redmine example it is not straightforward to migrate to new Redmine versions without breaking the EasyData plugin. More traditional black-box architectures, on the other hand, would only require that the web app provides a stable API, which is not always easy though. In this vein, the Hydra approach improves the decoupling of the LD consumer and the LD provider by providing a core vocabulary that can be used to describe and document generic Web APIs. The EasyData reverse engineering approach should follow a similar approach to be more general.

Compared to other solutions as Datalift, EasyData does not provide an interlinking technique to map heterogeneous metadata from different web sources. Instead, it facilitates a new way to enrich a legacy web app with links to external resources. These links can be discovered by means of any interlinking tool. EasyData does not replace interlinking tools, but makes it easier to integrate their outputs in the reengineered LOD application.

6. Conclusion

Reverse engineering can facilitate third parties to access Linked Open Data that emanate from legacy applications. The LOD extension approach presented in this paper makes it more flexible to implement non-functional requirements, such as secure access control, which are relevant for web applications engineering. The procedure is feasible as long as the application architecture is based on the MVC pattern and its source code is available. The LOD extension approach can be practiced at diverse layers of the architectural components of the application. It enables enriching the web views with RDFa and microdata, to reveal the application data binding model, to provide enriched web service descriptions and to have access to custom ReST-based API implementations on the controller layer.

The LOD model of a legacy web app can be disclosed, aligned with standard vocabularies and then exposed. Simple internal configurations make it possible to solve model alignment issues for legacy applications. An extension to the API can be also configured to publish the internal application API as LD, as part of an automated and flexible reengineering process. This process is not based on additional middleware components that work as wrappers or adaptors, since it may reduce the reliability and maintainability, and increase the complexity, of the overall software architecture.

On the con side, the coupling of generated LOD extensions and the reengineered web app source code may affect the plugin implementation, which could make maintenance more difficult when the original web app source code changes. To overcome this issue, coupling would be improved by extending the EasyData tags with R2RML-like annotations, such as RML [7], to describe the mappings between heterogeneous data sources in a source-agnostic, more extensible language.

Configuring the application with external models and schemata beyond the legacy application data model is highly recommendable to interlink heterogeneous entities in the Web of data. As a future work, the EasyData implementation is planned to be augmented to connect the generated LOD extensions with existing tools that facilitate the LD interlinking process [30]. Another line of work is to extend the implementation of the EasyData approach to facilitate developing non-functional privacy restrictions for privacy-preserving data publishing [12], which is particularly interesting when dealing with distributed data mashups.

Appendix

A. Public evaluation data

The SonarQube analysis on all the tools and frameworks of this paper are publicly available in sonarcloud.io[7]. The analysis were executed with SonarQube scanner[8]. To extract the relevant metrics for this paper, the sonarcloud.io Web API[9] was used. A JSON output is obtained by means of simple scripts like that of figure 6. Then the JSON output is converted to CSV[10] to do the analysis on a regular spreadsheet.

```
HASH='...'
URL='https://sonarqube.com/api/measures/component?
     componentKey'
COMP=$1
METRICS='cognitive_complexity,functions,sqale_index
     ,vulnerabilities,duplicated_lines,bugs,
     reliability_remediation_effort,
     class_complexity,complexity,statements,ncloc,
     duplicated_blocks,security_remediation_effort,
     function_complexity,code_smells'
while read COMPKEY; do
  curl -s -u $HASH: $URL=${COMPKEY}\&pageSize=-1\&
     metricKeys=$METRICS |
  jq '.component | { id, key, qualifier, path,
     measure: .measures[] }'
done < $COMP.txt
```

Figure 6. Script to query the sonarcloud.io Web API to obtain relevant SonarQube metrics

The enumeration of all the software modules and packages that are included in the analysis for the more complex LD frameworks (i.e. Stanbol and Datalift) is listed below. As for the more simple tools (i.e. Triplify, HydraBundle and EasyData), all modules are included in the analysis. In the D2Rq case, all modules were included except `src/de/fuberlin/wiwiss/d2rq/server`.

A.1. Stanbol modules

```
apache-stanbol-data
org.apache.stanbol.commons.owl
org.apache.stanbol.commons.security.core
org.apache.stanbol.commons.security.usermanagement
org.apache.stanbol.commons.web.base
org.apache.stanbol.commons.web.base.jersey
org.apache.stanbol.commons.web.home
org.apache.stanbol.commons.web.rdfviewable.writer
org.apache.stanbol.commons.web.resources
org.apache.stanbol.commons.web.viewable
org.apache.stanbol.commons.web.viewable.writer
```

[7]https://sonarcloud.io/organizations/dodero-github/projects
[8]https://docs.sonarqube.org/display/SCAN/Analyzing+Source+Code
[9]https://sonarcloud.io/web_api/
[10]https://konklone.io/json/

A.2. Datalift modules

```
allegrograph-connector/src/java/org/datalift/allegrograph
core/src/java/org/datalift/core
core/src/java/org/datalift/core/i18n/jersey
core/src/java/org/datalift/core/i18n/web
core/src/java/org/datalift/core/rdf
core/src/java/org/datalift/core/security
core/src/java/org/datalift/core/security/shiro
core/src/java/org/datalift/core/util
core/src/java/org/datalift/core/util/web
data2ontology/src/java/org/datalift/owl
data2ontology/src/java/org/datalift/owl/mapper
database-directmapper/src/java/net/antidot/semantic/rdf/
    model/tools
database-directmapper/src/java/net/antidot/semantic/rdf/
    rdb2rdf/commons
database-directmapper/src/java/net/antidot/semantic/rdf/
    rdb2rdf/dm/core
database-directmapper/src/java/net/antidot/semantic/rdf/
    rdb2rdf/main
database-directmapper/src/java/net/antidot/semantic/xmls/xsd
database-directmapper/src/java/net/antidot/sql/model/core
database-directmapper/src/java/net/antidot/sql/model/db
database-directmapper/src/java/net/antidot/sql/model/tools
database-directmapper/src/java/net/antidot/sql/model/type
database-directmapper/src/java/org/datalift/converter/
    dbdirectmapper
framework/src/java/org/datalift/fwk
framework/src/java/org/datalift/fwk/rdf
framework/src/java/org/datalift/fwk/rdf/json
framework/src/java/org/datalift/fwk/rdf/rio/rdfxml
framework/src/java/org/datalift/fwk/security
framework/src/java/org/datalift/fwk/util
framework/src/java/org/datalift/fwk/util/io
framework/src/java/org/datalift/fwk/util/web
framework/src/java/org/datalift/fwk/view
framework/tests/src/java/org/datalift/fwk/util
s4ac/src/java/org/datalift/s4ac
s4ac/src/java/org/datalift/s4ac/services
s4ac/src/java/org/datalift/s4ac/utils
stringtouri/src/java/org/datalift/stringtouri
```

References

[1] Aksac A, Ozturk O, Dogdu E (2012) A novel semantic web browser for user centric information retrieval: PERSON. Expert Systems with Applications 39(15):12,001–12,013

[2] Amundsen M (2014) APIs to affordances: A new paradigm for services on the web. In: Pautasso C, Wilde E, Alarcon R (eds) REST: Advanced Research Topics and Practical Applications, Springer, pp 91–106

[3] Auer S, Dietzold S, Lehmann J, Hellmann S, Aumueller D (2009) Triplify: light-weight linked data publication from relational databases. In: Proceedings of the 18th international conference on World Wide Web, pp 621–630

[4] Bizer C, Cyganiak R (2006) D2R Server – Publishing Relational Databases on the Semantic Web. In: Proceedings of the 5th International Semantic Web Conference, Athens, Georgia, USA

[5] Brooks FP (1995) The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition. Addison-Wesley Professional

[6] Caytiles RD, Lee S (2014) A Review of MVC Framework based Software Development. International Journal of Software Engineering and its Applications 8(10):213–220

[7] Dimou A, Vander-Sande M, Colpaert P, Verborgh R, Mannens E, de Walle RV (2014) RML: A generic language for integrated rdf mappings of heterogeneous data. In: Proceedings of the Workshop on Linked Data on the Web, 23rd International World Wide Web Conference

[8] Dotsika F (2010) Semantic APIs: Scaling up towards the Semantic Web. International Journal of Information Management 30(4):335–342

[9] Erling O (2007) Declaring RDF views of SQL data. In: W3C Workshop on RDF Access to Relational Databases

[10] Erling O, Mikhailov I (2009) RDF support in the Virtuoso DBMS. In: Pellegrini T, Auer S, Tochtermann K, Schaffert S (eds) Networked Knowledge - Networked Media. Integrating Knowledge Management, New Media Technologies and Semantic Systems, Studies in Computational Intelligence, vol 221, Springer, pp 8–24

[11] Fowler M (2002) Patterns of Enterprise Application Architecture. Addison-Wesley Longman, Boston, MA, USA

[12] Fung BCM, Wang K, Chen R, Yu PS (2010) Privacy-preserving data publishing: A survey of recent developments. ACM Computing Surveys 42(4):14:1–14:53

[13] Gill GK, Kemerer CF (1991) Cyclomatic complexity density and software maintenance productivity. IEEE Transactions on Software Engineering 17(12):1284–1288

[14] Groth P, Loizou A, Gray AJG, Goble C, Harland L, Pettifer S (2014) API-centric Linked Data integration: The OpenPHACTS Discovery Platform case study. Web Semantics: Science, Services and Agents on the World Wide Web 29:12–18

[15] Hausenblas M (2009) Exploiting Linked Data to Build Web Applications. IEEE Internet Computing 13(4):68–73

[16] Heath T, Bizer C (2011) Linked Data: Evolving the Web into a Global Data Space. Morgan & Claypool

[17] Hendler J, Shadbolt N, Hall W, Berners-Lee T, Weitzner D (2008) Web Science: an interdisciplinary approach to understanding the Web. Communications of the ACM 51(7):60–69

[18] Hyland B, Atemezing G, Villazón-Terrazas B (2014) Best practices for publishing linked data. Tech. Rep. TR/LDP, W3C

[19] Jentzsch A, Isele R, Bizer C (2010) Silk - generating rdf links while publishing or consuming linked data. In: Proceedings of the 2010 International Conference on Posters &#38; Demonstrations Track - Volume 658, pp 53–56

[20] Jöerg B, Ruiz-Rube I, Sicilia MA, Dvořák J, Jeffery K, Hoellrigl T, Rasmussen HS, Engfer A, Vestdam T, García-Barriocanal E (2012) Connecting closed world research information systems through the linked open data web. International Journal of Software Engineering and Knowledge Engineering 22(3):345–364

[21] Joksimovic S, Jovanovic J, Gasevic D, Zouaq A, Jeremic Z (2013) An empirical evaluation of ontology-based semantic annotators. In: Proceedings of the seventh international conference on Knowledge capture, ACM, pp 109–112

[22] Lanthaler M (2013) Creating 3rd Generation Web APIs with Hydra. In: Proceedings of the 22nd International Conference on World Wide Web, pp 35–38

[23] Lehmann J, Isele R, Jakob M, Jentzsch A, Kontokostas D, Mendes PN, Hellmann S, Morsey M, van Kleef P, Auer S, Bizer C (2015) DBpedia – A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. Semantic Web – Interoperability, Usability, Applicability 6(2):167–195

[24] McCabe TJ, Butler CW (1989) Design complexity measurement and testing. Communications of the ACM 32(12):1415–1425

[25] Oates BJ (2005) Researching information systems and computing. Sage

[26] Oren E, Delbru R, Catasta M, Cyganiak R, Stenzhorn H, Tummarello G (2008) Sindice.com: a document-oriented lookup in-

dex for open linked data. International Journal of Metadata, Semantics and Ontologies 3(1):37–52

[27] Oren E, Heitmann B, Decker S (2008) ActiveRDF: Embedding semantic web data into object-oriented languages. Web Semantics: Science, Services and Agents on the World Wide Web pp 191–202

[28] Pol K, Patil N, Patankar S, Das C (2008) A survey on web content mining and extraction of structured and semistructured data. In: Emerging Trends in Engineering and Technology, pp 543–546

[29] Prazeres CVS, Pimentel MG, Munson EV, Teixeira CA (2010) Toward semantic web services as MVC applications: from OWL-S via UML. Journal of Web Engineering 9(3):243–265

[30] Rajabi E, Sicilia MA, Sanchez-Alonso S (2014) An empirical study on the evaluation of interlinking tools on the web of data. Journal of Information Science 40(5):637–648

[31] Ruiz-Rube I, Dodero JM, Colomo-Palacios R (2015) A framework for software process deployment and evaluation. Information and Software Technology 59(3):205–221

[32] Scharffe F, Atemezing G, Troncy R, Gandon F, Villata S, Bucher B, Hamdi F, Bihanic L, K´ep´eklian G, Cotton F, Euzenat J, Fan Z, Vandenbussche PY, Vatant B (2012) Enabling linked data publication with the Datalift platform. In: AAAI Workshop on the 26th conference on Artificial Intelligence, AAAI Publications, pp 25–30

[33] Sheth AP, Gomadam K, Lathem J (2007) SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups. IEEE Internet Computing 11(6):91–94,

[34] Spanos DE, Stavrou P, Mitrou N (2010) Bringing relational databases into the semantic web: A survey. Semantic Web - Interoperability, Usability, Applicability 3(2):169–209

[35] Vaishnavi VK, Kuechler W (2015) Design Science Research Methods and Patterns, 2nd edn. CRC Press

[36] Wang HH, Damljanovic D, Payne T, Gibbins N, Bontcheva K (2012) Transition of Legacy Systems to Semantic Enabled Application: TAO Method and Tools. Semantic Web – Interoperability, Usability, Applicability 3(2):157–168

[37] Wölger S, Siorpaes K, Bürger T, Simperl E, Thaler S, Hofer C (2011) A survey on data interlinking methods. Tech. Rep. 2011-03-31, Semantic Technology Institute