by a brief introduction to iSPARQL. Section 5 illustrates the simplicity of using EvoOnt and iSPARQL for some common software evolution analyses tasks. To close the paper, Section 6 presents our conclusions, limitations of our approach, and some insight into future work.

2 Related Work

In the following, we briefly summarize a selection of interesting studies in the field of software research and ontologies. Firstly, *Coogle* (Code Google) [15] is the predecessor of our iSPARQL approach presented in this paper. With Coogle we were able to measure the similarity between Java classes of different releases of software projects. A major difference between the two approaches is that iSPARQL does not operate on in-memory software models in Eclipse[4], but on OWL ontologies (*i.e.*, on a well-established Semantic Web format). Furthermore, while in Coogle the range of usable similarity measures is limited to tree algorithms, the range of possible measures in iSPARQL includes all the measures from *SimPack*[5], our generic Java library of similarity measures for the use in ontologies.

Highly related to our approach is the work of Hyland-Wood *et al.* [10]. In their studies, the authors present an OWL ontology of software engineering concepts (*SEC*) including classes, tests, metrics, and requirements. Their ontology, however, does not include versioning information and data obtained from bug tracking systems (as in our ontology models). The structure of SEC is very similar to the language structure of Java. Note that our software ontology is based on *FAMIX* [4] that is a programming language-independent model for representing object-oriented source code, and thus, is able to represent software projects written in different programming languages.

D'Ambros and Lanza [3] present a visualization technique to uncover the relationships between data from a versioning and bug tracking system of a software project. To achieve this goal, they utilize the Release History Database (RHDB) introduced by Fischer *et al.* in [6].

Mäntylä *et al.* [13] and Shatnawi and Li [16] carry out an investigation of bad code smells in object-oriented software. While the first study additionally presents a taxonomy (in our sense an ontology) of smells and examines its correlations, both studies provide empirical evidence that some code smells can be linked with errors in software design.

Happel *et al.* [9] present their *KOntoR* approach that aims at storing and querying meta-data about software artifacts to foster software reuse. The software components are stored in a central repository. Furthermore, various ontologies for providing background knowledge about the components, such as the programming language and licensing models are proposed. It is certainly reasonable to integrate their models with ours in the future to result in an even larger fact base used to analyze large software systems.

Finally, Dietrich and Elgar [5] present an approach to automatically detect design patterns in Java programs based on an OWL design patterns ontology. Again, we think it would make sense to use their approach and ontology model to collect even more information about software projects. This would allow us to conduct further evaluations to measure the quality of software.

3 Software Ontology Models

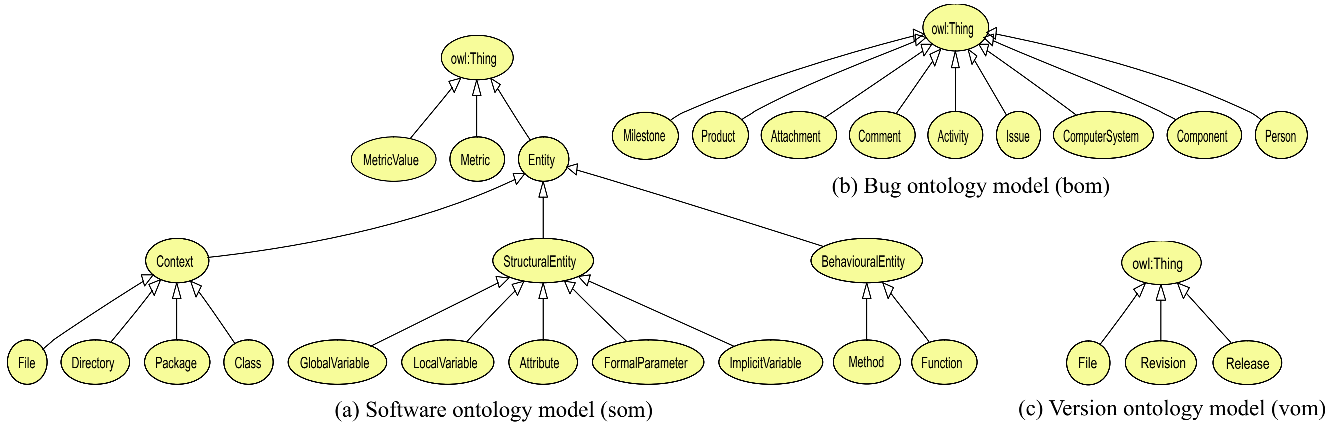
In this section, we describe our OWL software ontology models shown in Figure 1. We created three different models which encapsulate different aspects of object-oriented software source code: the *software ontology model (som)*, the *bug ontology model (bom)*, and the *version ontology model (vom)*. These models not only reflect the design and architecture of the software, but also capture information gathered over time (*i.e.*, during the whole life cycle of the software project). Such meta-data includes information about revisions, releases, and bug reports (among others).

3.1 Software Ontology Model

Our software ontology model (som) is based on *FAMIX* (FAMOOS Information Exchange Model) [4, 17]. FAMIX is a programming language-independent model for representing object-oriented source code. On the top level, the ontology specifies `Entity` that is the common superclass of all other entities, such as `BehaviouralEntity` and `StructuralEntity` (see Figure 1(a)). The sole parent of `Entity` is `owl:Thing` that is the implicit superclass of all user-defined classes in OWL ontologies. A `BehaviouralEntity` "represents the definition in source code of a behavioural abstraction, *i.e.*, an abstraction that denotes an action rather than a part of the state" (achieved by a method or function). A "`StructuralEntity`, in contrast, represents the definition in source code of a structural entity, *i.e.*, it denotes an aspect of the state of a system" [4] (*e.g.*, variable or parameter).

Figure 2 shows the original Famix model that we used as the basis for our model. When designing our OWL ontology, however, we made some changes to the original model: first, we introduced the three new classes `Context`, `File`, and `Directory`, the first one being the superclass of the latter ones. `Context` is designed as a container class to model the context in which a source code entity appears. A `File` is linked with a `Revision` of the version ontology (as described in Section 3.3) via the `isFileForRevision` property that is

[4] http://www.eclipse.org
[5] http://www.ifi.unizh.ch/ddis/simpack.html



(a) Software ontology model (som)

(b) Bug ontology model (bom)

(c) Version ontology model (vom)

Figure 1. The figure depicts the OWL class hierarchy (is-a) of the three created ontology models.

defined in the software ontology. This way it is possible to receive further information about the revisions of the file. Furthermore, due to the nature of OWL, we were able to omit the explicit modeling of association classes by adding new OWL object properties. For instance, to capture a method accessing a variable, the property `accesses` with domain `BehaviouralEntity` and range `StructuralEntity` can be used. Please note that we never used `Argument` and its subentities from the original FAMIX model. `Argument` further defines the parameters of an invocation (*e.g.*, a method invocation) since we found that the information about the expected arguments of a `BehaviouralEntity` was sufficient in all our experiments.

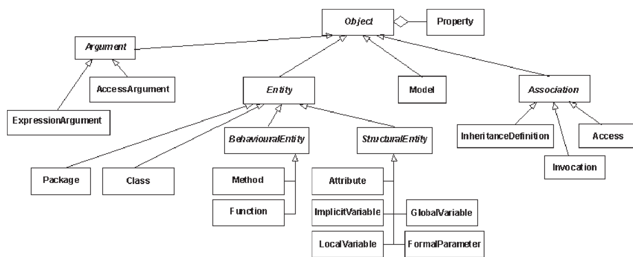


Figure 2. Original Famix model from [4].

In addition to the original model, we added the concept of software metrics to our ontology model. The class `Metric` defines an object-oriented source code metric as defined in [11]. An `Entity` can be connected to multiple metrics to measure various aspects of the design of the software component. This approach of integrating metrics into our model allows us to represent object-oriented metrics in an extendable way and to use the values of the metrics directly in our experiments (see Section 5).

3.2 Bug Ontology Model

Our bug ontology model (bom) is inspired by the bug tracking system *Bugzilla*[6]. The model is very shallow and defines nine OWL classes on the top level (see Figure 1(b)). `Issue` is the main class for specifying bug reports. It is connected to `Person` that is the class to model information about who reported the bug, and also to `Activity` that links additional details about the current status of the bug.[7] `Issue` has a connection to `Revision` (Section 3.3) via the `isResolvedIn` property. This way, information can be modeled about which revision successfully resolved a particular bug, and, vice versa, which bug reports were issued for a specific source code entity.

3.3 Version Ontology Model

The goal of our version ontology (vom) is to model the relationships between files, releases, and revisions of software projects. To that end, we defined the three OWL classes `File`, `Release`, and `Revision` (see Figure 1(c)) as well as the necessary properties to link these classes. For example, a `File` has a number of revisions and, therefore, is connected to `Revision` by the `hasRevision` property. At some point in time, developers of a software project usually decide to publish a new release of the software, which includes all the file revisions made until that point. In our model, this is reflected by the `isReleaseOf` property that relates Release and Revision.

[6] `http://www.bugzilla.org/`

[7] `https://bugs.eclipse.org/bugs` shows various concrete examples.

4 Our Approach: iSPARQL

This section succinctly introduces the relevant features of our iSPARQL framework that serves as the technical foundation to all experiments.[8] iSPARQL is an extension of SPARQL [14] that allows to query by triple patterns, conjunctions, disjunctions, and optional patterns. iSPARQL extends the official W3C SPARQL grammar but does not make use of additional keywords. Instead, iSPARQL introduces the idea of *virtual triples*. Virtual triples are not matched against the underlying ontology graph, but used to configure *similarity joins* [1]: they specify which pair(s) of variables (that are bound to resources with SPARQL) should be joined and compared using which type of similarity measure. Thus, they establish a *virtual relationship* between the resources bound to the variables describing their similarity. A similarity ontology defines the admissible virtual triples and links the different measures to their actual implementation in SimPack – our library of similarity measures. The similarity ontology also allows the specification of more complicated combinations of similarity measures, which we will call *similarity strategies* (or simply *strategies*) in the remainder of the paper. The next two sections briefly discuss the iSPARQL grammar and introduce some of the similarity strategies employed in the evaluation.

4.1 The iSPARQL Grammar

The relevant additional grammar statements are explained with the help of the example query shown in Listing 1. This query aims at comparing the same Java class from two releases of the `org.eclipse.compare` plug-in for Eclipse (used in all our experiments) by computing the structural difference of the classes (achieved by the "TreeEditDistance" measure, see Section 4.2).

```
1  PREFIX isparql: <java:ch.unizh.ifi.isparql.query.property.>
2  PREFIX som: <http://semweb.ivx.ch/software/som#>
3  PREFIX vom: <http://semweb.ivx.ch/software/vom#>
4
5  SELECT ?similarity
6  WHERE {
7   ?release1 vom:name "R3_1" .
8   ?release2 vom:name "R3_2" .
9
10  ?file1 som:hasRelease ?release1 .
11  ?file2 som:hasRelease ?release2 .
12  ?file1 som:uniqueName "org.eclipse.compare.MergeMessages.java" .
13  ?file2 som:uniqueName "org.eclipse.compare.MergeMessages.java" .
14  ?file1 som:hasClass ?class1 .
15  ?file2 som:hasClass ?class2 .
16  ?class1 som:uniqueName ?uniqueName1 .
17  ?class2 som:uniqueName ?uniqueName2 .
18
19  # ImpreciseBlockOfTriples (lines 21--26)
20
21  # NameStatment
22  ?strategy isparql:name ``TreeEditDistance'' .
23  # ArgumentStatment
24  ?strategy isparql:arguments (?class1 ?class2) .
25  # SimilarityStatement
26  ?strategy isparql:similarity ?similarity
27 }
```

Listing 1. Example iSPARQL query.

[8] An online demonstration of iSPARQL is available at `http://www.ifi.unizh.ch/ddis/isparql.html`

Strategy	Explanation
Jaccard measure (simple)	Set similarity between sets of methods/attributes of classes: determine the proximity of classes in terms of common and distinct methods/attributes [2, 8].
Levenshtein measure (simple)	String similarity between, for instance, class/method names: *Levenshtein* string edit distance to measure the relatedness of two strings in terms of the number of insert, remove, and replacement operations to transform one string into another string [12].
TreeEdit-Distance measure (simple)	Tree similarity between tree representations of classes: measuring the number of steps it takes to transform one tree into another tree by applying a set of elementary edit operations: insertion, substitution, and deletion of nodes [15].
Graph measure (simple)	Graph similarity between graph representations of classes: the measure aims at finding the maximum common subgraph (MCS) of two input graphs [18]. Based on the MCS the similarity between both input graphs is calculated.
Custom-Class-Measure (engineered)	User-defined Java class similarity measure: determines the affinity of classes by comparing their sets of method/attribute names. The names are compared by the Levenshtein string similarity measure. Individual similarity scores are weighted and accumulated to an overall similarity value.

Table 1. Selection of six iSPARQL similarity strategies.

In order to implement our virtual triple approach, we added an `ImpreciseBlockOfTriples` symbol to the standard SPARQL grammar expression of `FilteredBasicGraphPattern` [14]. Instead of matching patterns in the RDF graph, the triples in an `ImpreciseBlockOfTriples` act as *virtual triple* patterns, which are interpreted by iSPARQL's query processor.

An `ImpreciseBlockOfTriples` requires at least a `NameStatement` (line 22) specifying the similarity strategy and an `ArgumentsStatement` (line 24) specifying the resources under comparison to the iSPARQL framework. Note that iSPARQL also supports *aggregation strategies* – strategies which aggregate previously computed similarity scores of multiple similarity measures to an overall similarity value (not shown in the example). We found aggregators to be useful to construct overall (sometimes complex) similarity scores based on two or more previously computed similarity values. Finally, the `SimilarityStatement` (line 26) triggers the computation of the similarity measure with the given input arguments and delivers the result back to the query engine.

4.2 Similarity Strategies

Currently, iSPARQL supports all of the about 40 similarity measures implemented in SimPack. The reference to the implementing class as well as all necessary parameters are listed in the iSPARQL ontology. It is beyond the scope of this paper to present a complete list of implemented strategies. Therefore, Table 1 summarizes the five similarity strategies that we used to test the performance of iSPARQL on the `org.eclipse.compare` plug-in for Eclipse (Section 5). We distinguish between *simple* and *engineered* strategies: simple strategies employ a single,

atomic similarity measure of SimPack, whereas engineered strategies are a (weighted) combination of individual similarity measures whose resulting similarity scores get aggregated by a user-defined aggregator.

5 Experimental Results

We conducted four sets of experiments: (1) *code evolution measurements*: visualizing changes between different releases; (2) *refactoring experiments*: evaluation of the applicability of our iSPARQL framework to detect bad code smells; (3) *metrics experiments*: evaluation of the ability to calculate software design metrics; and (4) *ontological reasoning experiments*: investigation of the reasoning power within our software ontology models.

5.1 Experimental Setup and Data Sets

For our experiments, we examined 206 releases of the `org.eclipse.compare` plug-in for Eclipse. To generate an OWL data file of a particular release, it is first automatically retrieved from Eclipse's CVS repository and loaded into an in-memory version of our software ontology model, before it gets exported to an OWL file. To get the data from CVS and to fill our version ontology model, the contents of the Release History Database (RHDB) [6] for the compare plug-in are loaded into memory and, again, parsed and exported to OWL according to our version ontology model. While parsing the CVS data, the commit message of each revision of a file is inspected and searched for bug IDs. If a bug is mentioned in the commit message as, for instance, in *"fixed #67888: [accessibility] Go To Next Difference stops working on reuse of editor"*, the information about the bug is (automatically) retrieved from the web and also stored in memory. Finally, the data of the in-memory bug ontology model is exported to OWL.

5.2 Code Evolution Visualization

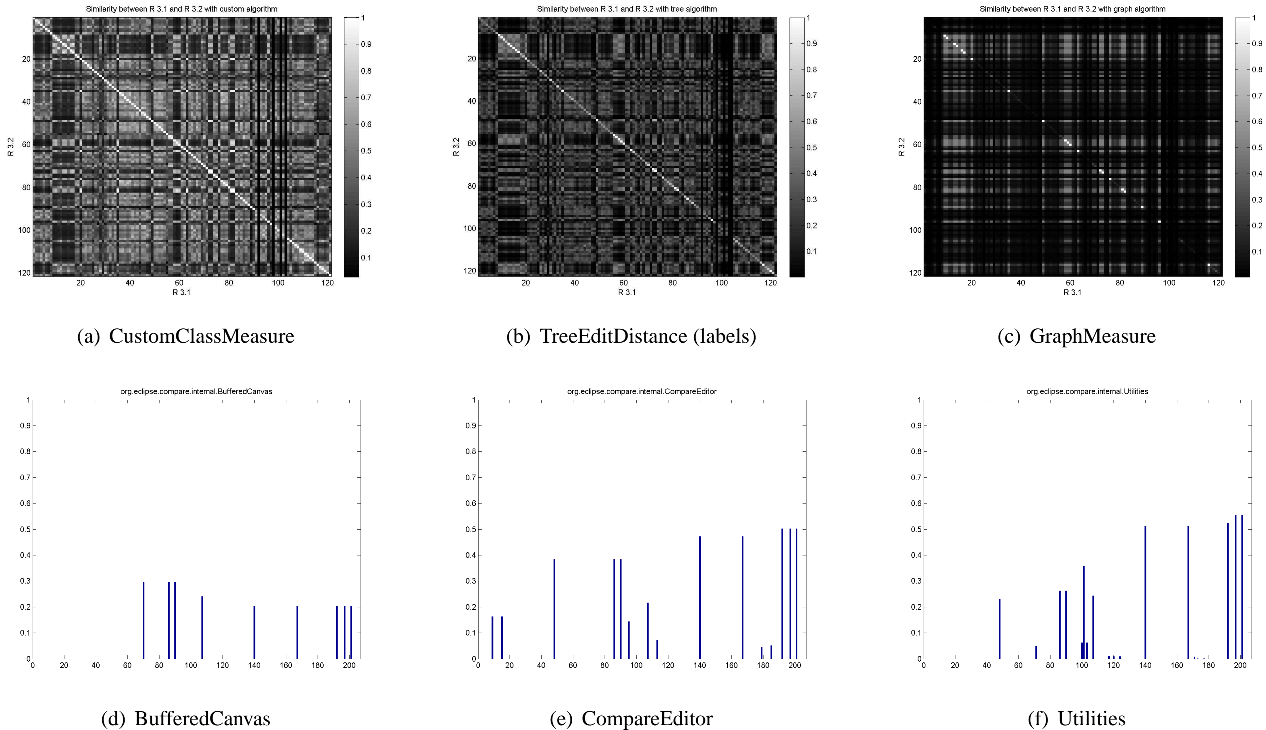
With the first set of experiments, we wanted to evaluate the applicability of our iSPARQL approach to the task of software evolution visualization (*i.e.*, the graphical visualization of changes in the code for a certain period of the life cycle of the software project). To that end, we compared all the Java classes of one major release with all the classes from another major release with different similarity strategies. Listing 1 (Section 4.1) shows the corresponding query for a particular class and the TreeEditDistance measure. The results for the releases R3_1 and R3_2 are shown in Figure 3. The heatmaps mirror the class code changes between the two releases of the project by using different colors for different similarity scores. Analyzing the generated heatmaps, we found out that the specialized CustomClassMeasure performed best for the given task. The combination of method/attribute set comparisons together with the Levenshtein string similarity measure for method/attribute names (Figure 3(b)) turned out to be less precise. Finally, the GraphMeasure (Figure 3(c)) was the least accurate indicator for the similarity of classes.

To shed some light on the history of a single Java class, we measured the similarity of the class from one release and the (immediate) next release and repeated this process for all releases and all classes. This resulted in an array of values $sim_{class}^{R_i,R_j}$ each of them expressing the similarity of the same class of two different releases. However, to visualize the *amount of change*, we plotted the inverse (*i.e.*, $1 - sim_{class}^{R_i,R_j}$) as illustrated in Figures 3(d–f) that show the history of changes of four distinct classes of the project. There are classes as, for example, `BufferedCanvas` which tend to have fewer changes as the project evolves over time. Other classes, such as `CompareEditor` (Figure 3(e)) are altered again and again, probably implying some design flaws or code smells. Then again, there are classes which tend to have more changes over time as shown in Figure 3(f) for the class `Utilities`.

5.3 Detection of Bad Code Smells

In a second set of experiments, we evaluated the applicability of our iSPARQL framework to the task of detecting bad code smells [7]. In other words, the question is whether iSPARQL is able to give you a *hint* that there *might* be a problem in the code. Can iSPARQL tell you if it could be solved, for instance, by refactoring current solutions? In order to solve this task, we selected two candidate smells, which we thought could be identified in the compare plug-in: *alien spider* anti-pattern and *long parameter list*. We succinctly present the results of our measurements.

The alien spider anti-pattern denotes the case, where many objects all mutually "know" each other, which is (1) bad object-oriented software design and (2) could lead to an uncomfortable situation when changes are made to a particular object, since, most probably, many other objects holding a reference to the changed object have to be modified too. We successfully identified the two-class version of the alien spider bad code smell in the compare plug-in by executing the iSPARQL query shown in Listing 2. The query returns a single result that states that the class `PatchWizard` uses a reference to the class `InputPatchPage` and vice versa. Inspecting class `PatchWizard`, one encounters the line addPage(fPatchWizardPage = **new** InputPatchPage(**this**)); expressing that a class `InputPatchPage` is instantiated and the instantiator (`PatchWizard`) is passed as reference. Supposing that some of the functionality of



(a) CustomClassMeasure

(b) TreeEditDistance (labels)

(c) GraphMeasure

(d) BufferedCanvas

(e) CompareEditor

(f) Utilities

Figure 3. The figure depicts the computed heatmaps of the between-version comparison of all classes of releases R3_1 and R3_2 using three different similarity strategies (Figures a–c). Furthermore, the history of changes for three distinct classes of the project is illustrated. Some classes tend to stabilize over time (Figure 3(d)), others are altered again and again (Figure 3(e)). Finally, there are classes which tend to have more changes as the project evolves (Figure 3(f)).

`PatchWizard` used by `InputPatchPage` is changed in the next release, `InputPatchPage` has to be adapted accordingly. Therefore, it could make sense to remove such mutual dependencies to overcome problems in the case the interface that a class exposes is changed.

```
1  PREFIX som: <http://semweb.ivx.ch/software/som#>
2
3  SELECT ?class1 ?class2
4  WHERE {
5   ?class1 som:hasAttribute ?var1 .
6   ?class2 som:hasAttribute ?var2 .
7   ?var2 som:hasDeclaredClass ?class1 .
8   ?var1 som:hasDeclaredClass ?class2 .
9   FILTER(?class1 != ?class2)
10 }
```

Listing 2. Alien spider query pattern.

Long method parameter lists are ugly, hard to understand, difficult to use, and chances are very high to change them over and over again [7]. In order to find the methods with long parameter lists in the Eclipse compare plug-in, we used the query shown in Listing 3. The 10 topmost results of the query are shown in Table 2. The method `merge` of the interface `IStreamMerger` takes nine parameters as input, and so does `TextStreamMerger`'s `merge` method since it implements `IStreamMerger`. These methods are possible candidates for a refactoring improving the overall design, usability, and quality of the software.

```
1  PREFIX som: <http://semweb.ivx.ch/software/som#>
2  PREFIX agg: <java:extensions.>
3
4  SELECT ?method ?parametercount
5  WHERE {
6   ?method som:hasFormalParameter ?formalParameter .
7   ?parametercount agg:countParameters ?method .
8   FILTER(?parametercount > 5)
9  } ORDER BY DESC(?parametercount)
```

Listing 3. Long parameter list query pattern.

5.4 Applying Software Metrics

With our third set of experiments, we wanted to demonstrate the possibility of calculating software design metrics with our iSPARQL system. Such metrics are explained in detail in [11]. For illustration purposes, we have chosen two of them which we will succinctly discuss in this section. Note that there is a close connection between bad code smells and metrics in the sense that metrics are often used to identify possible design flaws in object-oriented software

Class	Method	Method count
IStreamMerger	merge	9
TextStreamMerger	merge	9
CompareFilter	match	7
RangeDifferencer	createRangeDifference3	7
TextMergeViewer	mergingTokenDiff	7
TextMergeViewer-HeaderPainter	drawBevelRect	7
Differencer	findDifferences	6
Differencer	traverse	6
RangeDifferencer	rangeSpansEqual	6
WorkspacePatcher	readUnifiedDiff	6

Table 2. Results of long parameter list query pattern.

systems. To give a simple example, consider the query shown in Listing 4: the goal of this query is to detect possible *God classes* in the compare plug-in. A God class is defined as a class that potentially "knows" too much (its role in the program becomes all-encompassing), in our sense, has a lot of methods and instance variables. The query in Listing 4 calculates two metrics: NOM (number of methods) and NOA (number of attributes). Both metrics can be used as an indicator for possible God classes. The results are shown in Table 3. Having a look at class `TextMergeViewer`, one can see that the class is indeed very large with its 4344 lines of code. Also `CompareUIPlugin` is rather big with a total number of 1161 lines of code. Without examining the classes in more detail, we hypothesize that there might be some room for refactorings, possibly resulting in smaller, more easy to use classes.

```
1  PREFIX som: <http://semweb.ivx.ch/software/som#>
2  PREFIX agg: <java:extensions.>
3
4  SELECT ?GodClass ?NOM ?NOA
5  WHERE {
6   ?GodClass som:hasMethod ?Method .
7   ?NOM agg:countMethods ?GodClass .
8   FILTER(?NOM > 15) .
9   ?GodClass som:hasAttribute ?Attribute .
10  ?NOA som:countAttributes ?GodClass .
11  FILTER(?NOA > 15)
12 } ORDER BY DESC(?NOM)
```

Listing 4. God class query pattern.

God class	NOM	NOA
TextMergeViewer	115	91
CompareUIPlugin	46	42
ContentMergeViewer	44	36
CompareEditorInput	38	23
EditionSelectionDialog	30	26
CompareConfiguration	28	20
InputPatchPage	27	23
Diff	25	16
ComparePreferencePage	16	18

Table 3. Results of God class query pattern.

To support or discard our hypothesis, we measured the number of bug reports issued per class since one would assume a correlation between the number of class methods (attributes) and the number of tracked issues. To that end, we executed the query presented in Listing 5. Indeed, as the results in Table 4 clearly show, there is some correlation – the largest class in the project (`TextMergeViewer`) also has the largest number of filed bug reports.

```
1  PREFIX vom: <http://semweb.ivx.ch/software/vom#>
2  PREFIX bom:<http://semweb.ivx.ch/software/bom#>
3  PREFIX agg: <java:extensions.>
4
5  SELECT ?file ?issuecount
6  WHERE {
7   ?issue bom:isResolvedIn ?revision .
8   ?file vom:hasRevision ?revision .
9   ?issuecount agg:countIssues ?file
10 } ORDER BY DESC(?issuecount)
```

Listing 5. Bug reports query pattern.

Class	Bug reports count
TextMergeViewer	36
CompareEditor	16
Patcher	15
PreviewPatchPage	13
ResourceCompareInput	12
DiffTreeViewer	10
Utilities	10
CompareUIPlugin	9
StructureDiffViewer	9
PatchWizard	6

Table 4. Results of bug reports query pattern.

5.5 Ontological Reasoning

Last, with our final set of experiments, we aim at demonstrating the benefits of automated reasoning about facts, given our introduced OWL software ontology models. Since these models are specified in OWL-DL (Description Logic), we can apply corresponding reasoners, such as Pellet[9] or the Jena reasoners[10] to perform ontological reasoning. To give an example, we have chosen the query shown in Listing 6 that should find *orphan methods* (*i.e.*, methods that are not called by any other method in the project).

```
1  PREFIX som: <http://semweb.ivx.ch/software/som#> PREFIX inf:
2  <http://semweb.ivx.ch/software/inf#>
3
4  SELECT ?orphanMethod WHERE{
5   ?orphanMethod a inf:OrphanMethod
6  }
```

Listing 6. Orphan method query pattern.

To do so, we (1) defined the concept of an orphan method as depicted in Figure 4 in a separate ontology model, (2) loaded all ontology models into a Jena model with an external Pellet reasoner attached to it, and (3) executed the query against this model. The query returns numerous results of which we only present one of them here. It finds, for instance, the method `discardBuffer()` declared on `BufferedContent` that is never invoked by any other

[9] `http://www.mindswap.org/2003/pellet/`
[10] `http://jena.sourceforge.net/inference/`

class in the plug-in. Orphan methods could possibly be removed from the interface of a class without affecting the overall functionality of the system, resulting in cleaner and more easy to understand source code.

```
<owl:Class rdf:ID="Orphan">
 <rdfs:subClassOf rdf:resource="&som;Method"/>
 <owl:equivalentClass>
  <owl:Class>
   <owl:complementOf>
    <owl:Restriction>
     <owl:onProperty
        rdf:resource="&som;isInvokedBy"/>
     <owl:someValuesFrom
        rdf:resource="&som;BehaviouralEntity"/>
    </owl:Restriction>
   </owl:complementOf>
  </owl:Class>
 </owl:equivalentClass>
</owl:Class>
```

Figure 4. Orphan method OWL definition.

6 Conclusions, Limitations and Future Work

We presented a novel approach to mine semantically annotated software repositories. Based on the Semantic Web query language SPARQL, our iSPARQL framework together with EvoOnt provide the ability to mine software systems represented in the OWL data format. This format is principally used within the Semantic Web to share, integrate, and reason about data of various origin. We evaluated the use of this format in the context of analyzing the `org.eclipse.compare` plug-in for Eclipse.

To illustrate the power of using EvoOnt as a software evolution data exchange format, we conducted four sets of experiments in which we showed that iSPARQL and its imprecise querying facilities are indeed able to shed some light on the evolution of software systems. iSPARQL also helps to find bad code smells, and, thus, fosters refactoring. Furthermore, it enables the easy application of software design metrics to quantify the size and complexity of software, and, due to OWL's ontological reasoning support, allows to derive additional assertions (such as orphan methods) which are entailed from base facts.

A limitation of our approach is the loss of information due to the use of our FAMIX-based software ontology model. Language constructs, such *if-then-else* and *switch*-statements are not modeled in our ontology (neither are they in FAMIX). The effects are that measurements on the statements level of source code cannot be conducted.

Last, the current performance of our system is not satisfactory. Computing the heatmaps for even a small software project as the compare plug-in takes more than an hour (depending on the used similarity measure). Also, the amount of memory it takes to load and process the generated OWL data files is huge, almost exceeding the maximal available memory in our Java virtual machine (the amount of necessary memory is even larger if reasoning is turned on).

It is left to future work to analyze iSPARQL's applicability to other software analysis tasks as, for example, bug prediction. Furthermore, we think it makes sense to insert the results of some queries back into the model (*e.g.*, the results of metric computations), which would boost the performance of our approach. Also, we want to investigate different, more precise similarity measures to determine the affinity of Java classes (and other software entities). Given its support for similarity measures, iSPARQL allows the simple construction of instance-based machine learning operators as shown in [1]. We plan to extend iSPARQL with other analytic inference methods in the future. Such methods include *find deviation operators*, *find rule-based relationships*, *predict future behavior* (*e.g.*, bug prediction), and *cluster similar entities* in the data. This will further simplify the interpretation of the data. Coming back to the introductory example, iSPARQL seems to be a practical and easy to use tool to help Jane analyze her software project along a multitude of dimensions, making sure that she will perform brilliantly at her company's workshop.

References

[1] W. W. Cohen. Data Integration Using Similarity Joins and a Word-Based Information Representation Language. *ACM TOIS*, 18(3):288–321, 2000.

[2] W. W. Cohen, P. Ravikumar, and S. Fienberg. A Comparison of String Distance Metrics for Name-Matching Tasks. In *IIWeb Ws. at IJCAI '03*, 2003.

[3] M. D'Ambros and M. Lanza. Software Bugs and Evolution: A Visual Approach to Uncover Their Relationships. In *Proc. of the 10th Europ. Conf. on Softw. Maintenance and Reengineering (CSMR '06)*, pages 227–236, 2006.

[4] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - The FAMOOS Inf. Exchange Model. Technical report, University of Berne, Switzerland, 1999.

[5] J. Dietrich and C. Elgar. A Formal Description of Design Patterns Using OWL. In *Proc. of the 2005 Australian Software Engineering Conf. (ASWEC '05)*, Brisbane, Australia, 2005.

[6] M. Fischer, M. Pinzger, and H. Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proc. of the Int. Conf. on Softw. Maintenance (ICSM '03)*, pages 23–32, Amsterdam, Holland, 2003.

[7] M. Fowler. *Refactoring*. Addison-Wesley Longman, Amsterdam, 1999.

[8] P. Ganesan, H. Garcia-Molina, and J. Widom. Exploiting Hierarchical Domain Structure to Compute Similarity. *ACM TOIS*, 21(1):64–93, 2003.

[9] H.-J. Happel, A. Korthaus, S. Seedorf, and P. Tomczyk. KOntoR: An Ontology-enabled Approach to Software Reuse. In *Proc. of the 18th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE '06)*, San Francisco, CA, 2006.

[10] D. Hyland-Wood, D. Carrington, and S. Kapplan. Toward a Software Maintenance Methodology using Semantic Web Techniques. In *Proc. of the 2nd Int. IEEE Ws. on Software Evolvability at IEEE Int. Conf. on Software Maintenance (ICSM '06)*, pages 23–30, Philadelphia, PA, 2006.

[11] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer, Berlin, 2006.

[12] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[13] M. Mäntylä, J. Vanhanen, and C. Lassenius. A Taxonomy and an Initial Empirical Study of Bad Smells in Code. In *Proc. of the Int. Conf. on Software Maintenance (ICSM '03)*, Washington, DC, 2003.

[14] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. Technical report, W3C, 2006.

[15] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer. Detecting Similar Java Classes Using Tree Algorithms. In *Proc. of the 2006 Int. Ws. on Mining Software Repositories (MRS '06)*, New York, NY, 2006.

[16] R. Shatnawi and W. Li. A Investigation of Bad Smells in Object-Oriented Design Code. In *Proc. of the 3rd Int. Conf. on Information Technology : New Generations (ITNG'06)*, Washington, DC, 2006.

[17] S. Tichelaar. FAMIX Java language plug-in 1.0. Technical report, University of Berne, Switzerland, 1999.

[18] G. Valiente. *Algorithms on Trees and Graphs*. Springer, Berlin, 2002.