# Analyzing Software with iSPARQL

Christoph Kiefer, Abraham Bernstein, Jonas Tappolet

Department of Informatics, University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland,
{kiefer,bernstein}@ifi.unizh.ch, jtappolet@access.unizh.ch

**Abstract.** One of the most important decisions researchers face when analyzing software systems is the choice of a proper data analysis/exchange format. In this paper, we present EvoOnt, a set of software ontologies and data exchange format based on OWL. EvoOnt models software design, release history information, and bug-tracking meta-data. Since OWL describes the semantics of the data, EvoOnt is (1) easily extendible, (2) comes with many existing tools, and (3) allows to derive assertions through its inherent Description Logic reasoning capabilities. We show the usefulness of EvoOnt in combination with iSPARQL – our SPARQL-based Semantic Web query engine containing similarity joins. Together, EvoOnt and iSPARQL can accomplish a sizable number of tasks sought in software analyzing, such as an assessment of the amount of change between releases, the computation of software design metrics, or the detection of code smells. In a series of experiments with a real-world Java project, we show that a number of software analysis tasks can be reduced to a simple iSPARQL query on an EvoOnt dataset.

## 1 Introduction

Imagine the following situation: Sara is a project manager who has to report on one of her legacy systems she successfully manages for almost 5 years. Sara's company wants to know about the project's current development activity, its defects, its number of users, and, most important, its future maintenance costs. Somehow, Sara feels a little uncomfortable because there is no easy to use tool to perform all of the analysis tasks for her legacy system.

This is a very typical situation in software analysis and, generally, in software development: people analyzing their software systems either drown in a sea of specialized, task-specific tools or find no tools whatsoever. Altogether, analyzing large software systems can be a cumbersome and complex task.

In recent years, the rise of the Semantic Web has brought new possibilities also for software engineers. In this paper, we present how our software evolution ontology *EvoOnt* together with some off-the-shelf Semantic Web tools and our special *iSPARQL* query engine can help Sara to resolve to the various software analysis tasks involved in preparing her presentation without having to write a single line of code. *EvoOnt* is a set of software ontologies and data exchange format based on OWL. It provides the means to store all elements necessary for

software analyses including the software design itself as well as its release and bug-tracking information. Given that OWL is a W3C recommendation, a myriad of tools allow its *immediate processing* in terms of visualization, editing, querying, and debugging avoiding the need to write code or use complicated command line tools. OWL enables handling of the data based on its *semantics*, which allows the simple extension of the data model while maintaining the functionality of existing tools. Furthermore, given OWL's Description Logic foundation, any Semantic Web engine allows to *derive additional assertions* in the code such as orphan methods (see Section 5.5), which are entailed from base facts.

To complement EvoOnt, we developed our iSPARQL engine that extends the Semantic Web query language SPARQL with facilities to query for *similar* software entities (classes, methods, fields, etc.) in an EvoOnt dataset. Using a library of over 40 similarity measures, iSPARQL can exploit the semantic annotation of EvoOnt to compute statistical propositions about, for example, the evolution of software projects (see Section 5.2).

The remainder of this paper is structured as follows: next, we succinctly summarize the most important related work. Section 3 presents EvoOnt itself, which is followed by a brief introduction to iSPARQL. Section 5 illustrates the simplicity of using EvoOnt and iSPARQL for some common software evolution analysis tasks. To close the paper, Section 6 presents our conclusions, the 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 engineering research and the Semantic Web. Firstly, *Coogle* (Code Google) [13] 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[1], but on OWL ontologies (*i.e.*, on a well-established Semantic Web format). Furthermore, while in Coogle the range of applicable similarity measures is limited to tree algorithms, the range of possible measures in iSPARQL includes all the measures from *SimPack*, our generic Java library of similarity measures for the use in ontologies.[2]

Highly related to our approach is the work of Hyland-Wood *et al.* [8]. In their studies, the authors present an OWL ontology of software engineering concepts (SEC) including classes, tests, metrics, and requirements. Their ontology does, however, not include versioning information and data obtained from bug-tracking systems (as modeled in our ontologies). The structure of SEC is very similar to the language structure of Java. Note that our software ontology is based on FAMIX [2] that is a programming language-independent model to

[1] `http://www.eclipse.org`
[2] `http://www.ifi.unizh.ch/ddis/simpack.html`

represent object-oriented source code, and thus, is able to represent software projects written in different programming languages.

Both, Mäntylä *et al.* [11] and Shatnawi and Li [14] carry out an investigation of code smells in object-oriented software source code. 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.* [7] 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 presented. 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.

Dietrich and Elgar [3] present a technique 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.

Finally, we would like to point out that EvoOnt shares a lot of commonalities with Baetle[3] that is an ontology which focuses heavily on the information kept in bug databases, and that makes use of many other well-established Semantic Web ontologies, such as the Dublin Core[4] and FOAF[5].

## 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 project). Such meta-data includes information about revisions, releases, and bug reports.

### 3.1 Software Ontology Model

Our software ontology model (som) is based on *FAMIX* (FAMOOS Information Exchange Model) [2] that 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)). A `BehaviouralEntity` "represents the definition in source code of a behavioural abstraction,

[3] `http://code.google.com/p/baetle/`
[4] `http://dublincore.org/documents/dcq-rdf-xml/`
[5] `http://www.foaf-project.org/`

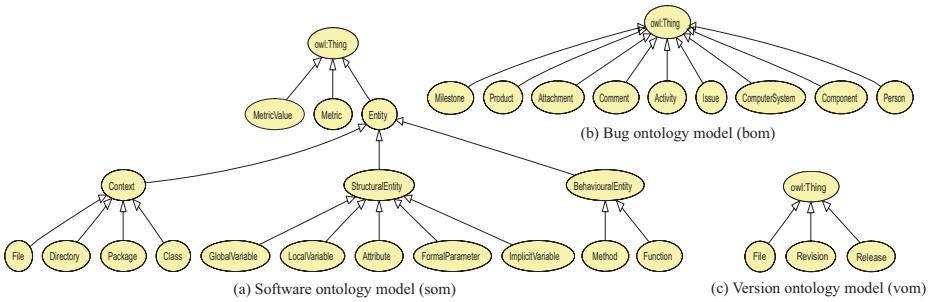



**Fig. 1.** The figure depicts the OWL class hierarchy (is-a) of the three composed ontology models. A larger version of the figure including object and data type properties is available at `http://www.ifi.unizh.ch/ddis/evoont.html`.

*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" [2] (*e.g.*, variable or parameter).

When designing our OWL ontology, we made, however, some changes to the original FAMIX model: first, we introduced the three new classes `Context`, `File`, and `Directory`, the first one being the superclass of the latter ones. `Context` is 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 (described in Section 3.3) via the `isFileForRevision` property that is 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` is defined.

Moreover, we added the concept of software metrics to our ontology model. The class `Metric` defines an object-oriented source code metric as defined in [9]. 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 extendible way and to use the values of the metrics directly in our experiments (see Section 5.4).

### 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 (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

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

of the bug.[7] `Issue` has a connection to `Revision` (see Section 3.3) via the `isResolvedIn` property. This way, information can be modeled about which file revision successfully resolved a particular bug, and vice versa, which bug reports were issued for a specific source code file.

### 3.3 Version Ontology Model

The goal of our version ontology model (vom) is to specify the relations between files, releases, and revisions of software projects. To that end, we defined the three OWL classes `File`, `Release`, and `Revision` (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, the developers of a software project usually decide to publish a new release, which includes all the revisions made until that point. In our model, this is reflected by the `isReleaseOf` property that relates `Release` and `Revision`.

## 4 Our Approach: iSPARQL

This section succinctly introduces the relevant features of our iSPARQL framework that serves as the technical foundation to all our experiments.[8] iSPARQL is an extension of SPARQL [12] that allows to query RDF graphs. It extends the official 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: 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 relation* between the resources bound to the variables describing the resource's 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.[9] Next, we briefly discuss the iSPARQL grammar and introduce some of the *similarity strategies* – complex combinations of similarity measures – employed in the evaluation.

### 4.1 The iSPARQL Grammar

The relevant additional grammar statements are explained with the help of the example query in Listing 1.1. This query aims at comparing two versions of a concrete Java class from two different releases of a software project by computing the structural difference of the classes (achieved by the TreeEditDistance measure, see Section 4.2).

[7] `https://bugs.eclipse.org/bugs` shows various concrete examples.
[8] A demonstration is available at `http://www.ifi.unizh.ch/ddis/isparql.html`.
[9] `http://www.ifi.unizh.ch/ddis/simpack.html`

```
1  PREFIX isparql: <java:arq.propertyfunction.>
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 20--25)
20  # NameStatment
21  ?strategy isparql:name ''TreeEditDistance'' .
22  # ArgumentStatment
23  ?strategy isparql:arguments (?class1 ?class2) .
24  # SimilarityStatement
25  ?strategy isparql:similarity ?similarity
26 } ORDER BY DESC (?similarity)
```

**Listing 1.1.** iSPARQL example query.

In order to implement our virtual triple approach, we added an `ImpreciseBlockOfTriples` symbol to the official W3C SPARQL grammar expression of `FilteredBasicGraphPattern`.[10] 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 21) specifying the similarity measure and an `ArgumentsStatement` (line 23) 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). Finally, the `SimilarityStatement` (line 25) 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 four similarity strategies that we use in Section 5. We distinguish between *simple* and *engineered* strategies: simple strategies employ a single, atomic similarity measure of SimPack, whereas engineered strategies

[10] Note that we refer to the W3C SPARQL working draft of October 4, 2006.

| Strategy | Explanation |
|---|---|
| Levenshtein measure (simple) | String similarity between, for instance, class/method names: *Levenshtein* string edit distance measuring the relatedness of two strings in terms of the number of insert, remove, and replacement operations to transform one string into another string [10]. |
| TreeEditDistance 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 [13]. |
| Graph measure (simple) | Graph similarity between graph representations of classes: the measure aims at finding the maximum common subgraph (MCS) of two input graphs [15]. Based on the MCS the similarity between both input graphs is calculated. |
| CustomClassMeasure (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 four iSPARQL similarity strategies.

are a (weighted) combination of individual similarity measures whose resulting similarity scores get aggregated by a user-defined aggregator.

## 5 Experimental Results

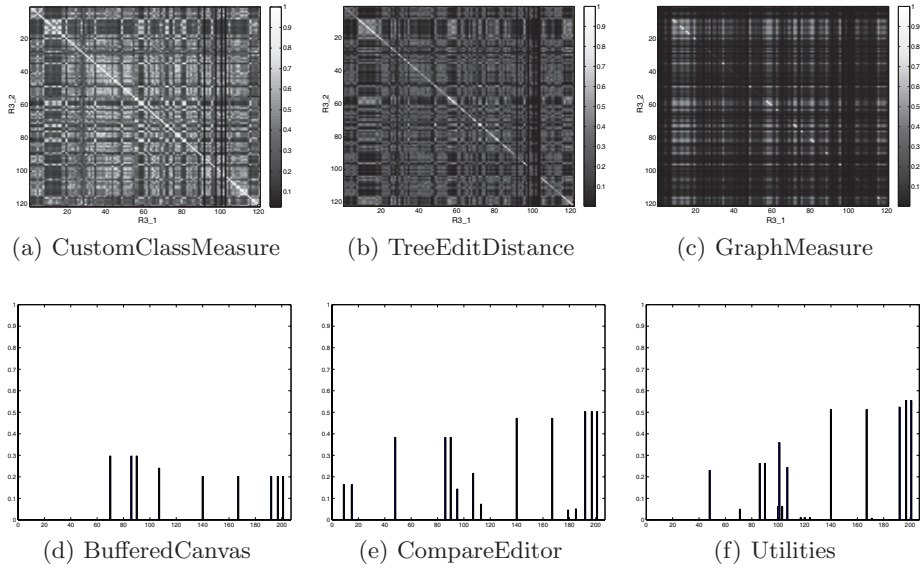
We conducted five sets of experiments: (1) *code evolution measurements*: visualizing changes between different releases; (2) *refactoring experiments*: evaluation of the applicability of our iSPARQL framework to detect code smells; (3) *metrics experiments*: evaluation of the ability to calculate software design metrics; (4) *ontological reasoning experiments*: investigation of the reasoning power within our software ontology models; and (5) *density measurements*: determining the amount of bug-fixing and "ordinary" software development as measured by all software engineering activities.

### 5.1 Experimental Setup and Datasets

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) [5] 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 Experiment 1: 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




(a) CustomClassMeasure (b) TreeEditDistance (c) GraphMeasure

(d) BufferedCanvas (e) CompareEditor (f) Utilities

**Fig. 2.** Figures 2(a–c) depict the computed heatmaps of the between-version comparison of all the classes of releases R3_1 and R3_2 using three different similarity strategies. Furthermore, the history of changes for three distinct classes of the project is illustrated in Figures 2(d–f).

graphical visualization of code changes for a certain time span in 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 1.1 (see 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 2. The heatmaps mirror the class code changes between the two releases of the project by using different shades of gray for different similarity scores in the interval $[0, 1]$. Analyzing the generated heatmaps, we found 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 2(b)) turned out to be less precise. In all our experiments, the GraphMeasure (Figure 2(c)) was the least accurate indicator for the similarity of classes.

Furthermore, 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 classes and releases. This resulted in an array of values $sim_{class}^{R_i,R_j}$, each value expressing the similarity of the same class of two different releases $R_i$ and $R_j$. However, to visualize the *amount of change*, we plotted the inverse (*i.e.*, $1 - sim_{class}^{R_i,R_j}$) as illustrated in Figures 2(d–f) that show the history of changes for three distinct classes of the project. There are classes

such as `BufferedCanvas` which tend to have fewer changes as the project evolves over time. Other classes such as `CompareEditor` (Figure 2(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 2(f) for the class `Utilities`.

### 5.3 Experiment 2: Detection of Code Smells

In a second set of experiments, we evaluated the applicability of our iSPARQL system to the task of detecting code smells [6]. 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 the current state? 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*.

The alien spider anti-pattern denotes the case where many objects all mutually "know" each other, which is, first, bad object-oriented software design; and second, could lead to an uncomfortable situation when changes are made to a particular object, because, 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 anti-pattern in the compare plug-in by executing the query shown in Listing 1.2. 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 a reference. Supposing that some of the functionality of `PatchWizard` used by `InputPatchPage` is changed in the future, `InputPatchPage` also has to be adapted. Therefore, it could make sense to remove such mutual dependencies to overcome problems when 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 1.2.** 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 [6]. In order to find the methods with long parameter lists in the Eclipse compare plug-in, we used the query given in Listing 1.3. The 10 topmost answers to the query are shown in Table 2. The method `merge` of the interface `IStreamMerger` takes

| Class | Method | Number of parameters |
|---|---|---|
| IStreamMerger | merge | 9 |
| TextStreamMerger | merge | 9 |
| CompareFilter | match | 7 |
| RangeDifferencer | createRangeDifference3 | 7 |
| TextMergeViewer | mergingTokenDiff | 7 |
| TextMergeViewerHeaderPainter | drawBevelRect | 7 |
| Differencer | findDifferences | 6 |
| Differencer | traverse | 6 |
| RangeDifferencer | rangeSpansEqual | 6 |
| WorkspacePatcher | readUnifiedDiff | 6 |

**Table 2.** Results of long parameter list query pattern.

nine parameters as input, and so does `TextStreamMerger`'s `merge` method because it implements `IStreamMerger`. These methods are possible candidates for a refactoring to improve the overall design, usability, and quality of the software.

```
1 PREFIX som: <http://semweb.ivx.ch/software/som#>
2 PREFIX agg: <java:arq.propertyfunction.>
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 1.3.** Long parameter list query pattern.

### 5.4 Experiment 3: Applying Software Metrics

With our third set of experiments, we wanted to demonstrate the possibility of calculating software design metrics using our iSPARQL system. Such metrics are explained in detail in [9]. 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 code smells and metrics in the sense that metrics are often used to identify possible design flaws in object-oriented software systems.

```
 1 PREFIX som: <http://semweb.ivx.ch/software/som#>
 2 PREFIX agg: <java:arq.propertyfunction.>
 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 agg:countAttributes ?GodClass .
11  FILTER(?NOA > 15)
12 } ORDER BY DESC(?NOM)
```

**Listing 1.4.** God class query pattern.

To give a simple example, consider the query shown in Listing 1.4: 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

| 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.

| Buggy class | NOB |
|---|---|
| **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.

becomes all-encompassing), in our sense, has a lot of methods and instance variables. The query in Listing 1.4 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 closer look at class `TextMergeViewer`, one can observe 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, which possibly result in smaller and more easy to use classes.

To support or discard our hypothesis, we measured the number of bug reports issued per class, because we assume a correlation between the number of class methods (attributes) and the number of filed bug reports. To that end, we executed the query presented in Listing 1.5. Indeed, there is a correlation as the results in Table 4 clearly show: the two classes `TextMergeViewer` and `CompareUIPlugin` are also among the top 10 most buggiest classes in the project.

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

**Listing 1.5.** Bug reports query pattern.

### 5.5 Experiment 4: Ontological Reasoning

Our fourth set of experiments aims at demonstrating the benefits of automated reasoning about facts given our introduced OWL software ontology models. Because these models are specified in OWL-DL, we can apply the OWL reasoners from Jena[11] or the Pellet reasoner[12] to perform ontological reasoning.

To give an example, we have chosen the query shown in Listing 1.6 that should find *orphan methods* (*i.e.*, methods that are not called by any other method in the project). The query finds all `?orphanMethod`'s, gets any `isInvokedBy` and filters those which passed through the optional branch. This query requires ontological reasoning because `isInvokedBy` is defined as `owl:inverseOf invokes` in our software ontology model. In other words, results of the form *method1 isInvokedBy method2* must be inferred from the `invokes`-statements. To do so, we loaded the software ontology model into a Jena model with an OWLMicro reasoner attached to it and executed the query against this inference model.

The query returns numerous results of which we only present one of them here. It finds, for instance, the public method `discardBuffer()` declared on the class `BufferedContent`. This method is never invoked by any other 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 to result in a more clean and easy to understand source code.

```
1 PREFIX som: <http://semweb.ivx.ch/software/som#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3
4 SELECT ?orphanMethod
5 WHERE {
6  ?orphanMethod rdf:type som:Method .
7  OPTIONAL { ?orphanMethod som:isInvokedBy ?invoker }
8  FILTER ( !bound(?invoker) )
9 }
```

**Listing 1.6.** Orphan method query pattern.

### 5.6 Experiment 5: Defect and Evolution Density

With our final set of experiments, we aim at determining a file's as well as a whole software project's *Defect* and *Evolution Density*. Note that in this context, we consider files as "containers" for classes and instance variables (*i.e.*, they may contain multiple classes as well as inner classes). Inspired by Fenton [4], we define the defect density $DED_f$ of a file $f$ as

$$DED_f = \frac{NOB}{NOR} \tag{1}$$

where NOR is the number of revisions of a file in a versioning system (NOR's query pattern is omitted here since it looks very similar to NOB's in Section 5.4).

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

| Class | NOR | NOB | EVD | DED |
|---|---|---|---|---|
| StatusLineContributionItem | 3 | 3 | 0.000 | 1.000 |
| CompareNavigator | 3 | 2 | 0.333 | 0.667 |
| IResourceProvider | 4 | 2 | 0.500 | 0.500 |
| DifferencesIterator | 10 | 5 | 0.500 | 0.500 |
| PatchProjectDiffNode | 2 | 1 | 0.500 | 0.500 |
| IStructureCreator | 11 | 4 | 0.636 | 0.364 |
| PreviewPatchPage | 37 | 13 | 0.649 | 0.351 |
| UnmatchedHunkTypedElement | 3 | 1 | 0.667 | 0.333 |
| WorkerJob | 3 | 1 | 0.667 | 0.333 |
| ResourceCompareInput | 38 | 12 | 0.684 | 0.316 |
| Patcher | 51 | 15 | 0.706 | 0.294 |
| CompareEditor | 57 | 16 | 0.719 | 0.281 |
| RangeDifference | 11 | 3 | 0.727 | 0.273 |
| ResizableDialog | 11 | 3 | 0.727 | 0.273 |
| WorkspacePatcher | 11 | 3 | 0.727 | 0.273 |

**Table 5.** Evolution and defect density of the org.eclipse.compare plug-in.

In other words, Equation 1 computes the ratio of the number of bug reports over the total number of revisions of a file $f$.

Next, we define a file's/project's *Evolution Density* as counterpart to defect density. When we refer to evolution density, we think of all the changes made to a software system which were not bug-fixing, but "ordinary" software development, such as functional extension and improvement, adaption, and testing. The evolution density $EVD_f$ of a file $f$ is defined as:

$$EVD_f = 1 - DED_f \tag{2}$$

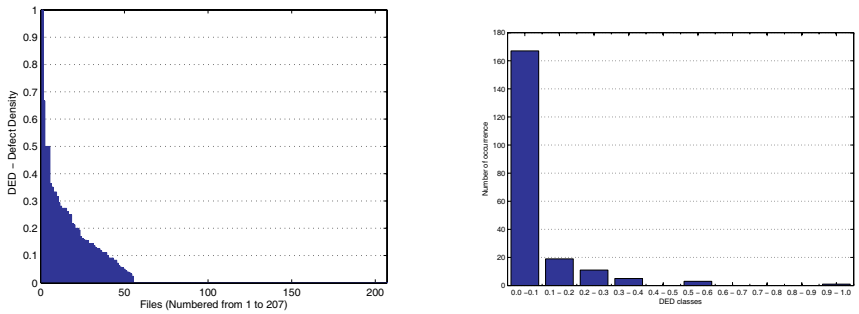
Table 5 lists evolution and defect density for the 15 topmost classes of the `org.eclipse.compare` plug-in in descending order of defect density. Visualizing the defect density (Figure 3(a)) brings to light some interesting facts: first, only about 25% of all source files contain bugs at all. Nearly 75% of the code is free of defects (measured by the reported bugs); second, the concentration of the errors is exponentially decreasing (*i.e.*, only few files have a high concentration of bugs). This is further illustrated in Figure 3(b) that shows a histogram of the number of classes in the project per 0.1 DED interval.

Finally, to calculate measures *over all software engineering activities* in the project, *Total Evolution Density* ($TEVD$) and *Total Defect Density* ($TDED$) are defined as shown in Equations 3 and 4:

$$TEVD = \frac{\sum_{f=1}^{n} EVD_f}{n} \tag{3}$$

$$TDED = \frac{\sum_{f=1}^{n} DED_f}{n} = 1 - TEVD \tag{4}$$

For the `org.eclipse.compare` plug-in R3_2_1, the value for $TDED$ is 0.054, which expresses that 5.4% of all activities in the project is due to bug-fixing and 94.6% due to functional extension etc. These findings seem to contradict those of Boehm [1] who found that about 12% of all software engineering tasks are




(a) DED per file. (b) DED histogram.

**Fig. 3.** The figure shows the defect density *DED* per file and the number of classes per 0.1 DED interval in the `org.eclipse.compare` plug-in R3_2_1.

bug-fixing. We hypothesize that the *time span* of the measurements and the *bug reporting discipline* (among others) are reasons for this divergence in results and postpone it to future work to prove or reject this hypothesis.

## 6 Conclusions, Limitations, and Future Work

In this paper, we presented a novel approach to analyze software systems using Semantic Web technologies. Based on the Semantic Web query language SPARQL, our iSPARQL framework together with EvoOnt provide the ability to examine software represented in OWL. 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 and iSPARQL, we conducted five sets of experiments in which we showed, first, that iSPARQL and its imprecise querying facilities are indeed able to shed some light on the evolution of software systems; second, that iSPARQL also helps to find code smells, hence, fosters refactoring; third, that it enables the easy application of software design metrics to quantify the size and complexity of software; forth, that it, due to OWL's ontological reasoning support, furthermore allows to derive additional assertions, which are entailed from base facts; and fifth, that it enables defect and evolution density measurements expressing the amount of bug-fixing and "ordinary" software development as measured by all software engineering tasks.

A limitation of our approach is the loss of information due to the use of our FAMIX-based software ontology model. Language constructs such as switch-statements are not modeled in our ontology. 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 employed 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 memory load is even larger if reasoning is turned on).

It is left to future work to analyze iSPARQL's applicability to other software analysis tasks, such as bug prediction. We also think it makes sense to insert the results of some queries back into the model (*e.g.*, the results of some metric computations), which would further boost the performance of our approach. Also, we want to investigate different, more precise similarity measures to determine the affinity of software entities. Coming back to the introductory example, iSPARQL seems to be a practical and easy to use tool to help Sara analyze her software project along a multitude of dimensions, making sure that she will neither drown in a sea of diffusion nor be criticized by her company's executives.

## References


1. B. W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.
2. S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - The FAMOOS Inf. Exchange Model. Technical report, University of Berne, Switzerland, 1999.
3. J. Dietrich and C. Elgar. A Formal Description of Design Patterns Using OWL. In *Proc. of the 2005 Australian Software Engineering Conf.*, Brisbane, Australia, 2005.
4. N. E. Fenton. *Software Metrics: A Rigorous Approach*. Int. T. Comp. Press, 1991.
5. 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 Software Maintenance*, pages 23–32, Amsterdam, Netherlands, 2003.
6. M. Fowler. *Refactoring*. Addison-Wesley, 1999.
7. 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*, San Francisco, CA, 2006.
8. 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 ICSM '06*, pages 23–30, Philadelphia, PA, 2006.
9. M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.
10. V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707–710, 1966.
11. 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*, Washington, DC, 2003.
12. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. Technical report, W3C, 2006.
13. 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*, New York, NY, 2006.
14. 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*, Washington, DC, 2006.
15. G. Valiente. *Algorithms on Trees and Graphs*. Springer, 2002.