Extracting knowledge from software artefacts to assist software project stakeholders

Miriam Gil
Departament d'Informàtica, Universitat de València
Burjassot, Spain *miriam.gil@uv.es*

Victoria Torres
PROS Research Center, Universitat Politècnica de València
València, Spain *vtorres@pros.upv.es*

Manoli Albert
PROS Research Center, Universitat Politècnica de València
València, Spain *malbert@pros.upv.es*

Vicente Pelechano
PROS Research Center, Universitat Politècnica de València
València, Spain *pele@pros.upv.es*

Abstract

Software development methods should foster the exploitation of artefacts from existing code bases in order to improve software development productivity. These artefacts are commonly stored in repositories from which extracting knowledge is very difficult for several reasons, i.e., the stored data is represented in a wide variety of formats or is not usually linked properly to all the related artefacts. In this work, we address the challenge of extracting knowledge from different artefacts that can be produced within a software project. To this end, we present a Persistent Knowledge Monitor (PKM) for handling several kinds of knowledge and information related to a software project. The PKM bases on the JSON format to structure and store the different artefacts. By using a common representation format, we are able to extract knowledge more easily. Also, we provide a query language for searching and reasoning on the stored data.

Keywords: Software development methods, knowledge base, knowledge extraction, knowledge retrieval, software artefacts

1. Introduction

Software development methods (hereafter SDM) are commonly designed to support stakeholders along the whole software lifecycle, i.e., from creation to delivery. SDM provide a common framework for software companies and organizations to deliver software in a structured and methodological way. However, to improve software development productivity, SDM should foster the reuse of code and software artefacts from existing code bases [9]. There is an important amount of information that can be extracted from artefacts developed in a software project. This information can be used to better understand the own project.

Extracting knowledge from current repositories is difficult for many reasons. For example, the stored data from the different artefacts (i.e., source code, models, tests, requirements, etc.) are represented in a wide variety of formats (e.g., xml, xmi, json, java, c, etc.), are not usually linked properly to all the related artefacts to describe jointly a specific part of the system (e.g., chunks of source code related to the dynamic behaviour of an object described in a state diagram), and are represented in different abstraction levels (e.g., source code, UML class diagrams, and requirements are usually represented at low,

medium and high abstraction levels respectively). Therefore, processing data turns into a very complex and cumbersome process.

Existing research has mainly been focused on the use of topic models[1], association rules, and heuristics to mine source code repositories for application to traceability, extraction of tactics (a means of satisfying a quality-attribute-response measure by manipulating some aspect of a quality attribute mode), and monitoring changes in source code [5]. The mining of code repositories from the standpoint of knowledge extraction can help project stakeholders to improve their activities during software development. Project stakeholders need to cope with a massive amount of knowledge throughout the typical life cycle of modern projects. This knowledge includes expertise related to the software development phases (e.g., implementation) using a wide variety of methods and tools, including development methodologies (e.g., waterfall, agile), software tools (e.g., Eclipse), programming languages (e.g., Java, SQL), and deployment strategies (e.g., Docker, Jenkins). However, there is no explicit integration of these various types of knowledge with software development projects so that project stakeholders can avoid having to search over and over for similar and recurrent solutions to tasks and reuse this knowledge.

With the aim of mitigating the project stakeholders' effort to integrate into a software project, in this work we address the challenge of extracting knowledge from different artefacts that can be produced within a software development project. To this end, we present a Persistent Knowledge Monitor (PKM) for handling (i.e., storing, retrieving, merging and checking for consistency) several kinds of knowledge and information related to a software project. The PKM bases on the JSON format to structure and store the different artefacts that are produced during the development project. By using a common representation format to describe the artefacts developed in a software project, we are able to extract knowledge more easily. In order to obtain knowledge from the PKM, we provide a query language for searching and reasoning on the stored data or knowledge. With this approach, we expect an improvement from the management and transformation of informal data into material (herein called 'knowledge') that can be assimilated by any party involved in a development process. This work is being developed within the DECODER H2020 project (https://www.decoder-project.eu/) whose major objective is to provide powerful tools for project stakeholders to get thorough understanding of a given piece of software.

The remainder of the paper is structured as follows. Section 2 introduces an overview over existing work in the software development research field. Section 3 provides an overview over the PKM metamodel, describing its main components and the relationships among them. Section 4 presents the query language proposed in this work to extract knowledge of the PKM. Section 5 presents the experiment performed to evaluate the query language. Finally, Section 6 presents the conclusions.

2. State of the Art

Although knowledge is of great value within a software development project, SDM are not traditionally centered on it but instead of processes. More traditional approaches, like the Waterfall model or the V-model, facilitate knowledge sharing primarily through documentation. More modern development methodologies, following the agile mainstream, suggest that most of the written documentation can be replaced by enhanced informal communications among team members internally and between the team and the customers with a stronger emphasis on tacit knowledge rather than explicit knowledge [4]. In [3] the authors compare Agile and Tayloristic methods as approaches for Knowledge Sharing. This work points out the main drawbacks of the Tayloristic approach, which are that it does not address issues of how well users internalize explicit knowledge and the sharing of tacit knowledge that is not externalized. Also, the authors identify as relevant praxis in Agile methods the emphasis on people, communities of practice, communication, while collaboration excels in facilitating the practice of sharing tacit knowledge at a team level. DevOps [5], which is currently one of the most modern software development

[1] Statistical models that provide a means to automatically index, search, cluster, and structure unstructured and unlabeled documents.

approaches, leverages improved continuity and efficiency. Thus, the development phases are integrated much more tightly than in the standard linear or V-shaped processes, allowing knowledge to flow rapidly between phases. DevOps did not address representation formalism integration though.

In the literature, we can find different metamodels targeted to represent the knowledge that can be extracted from software artefacts. These include the Knowledge Discovery Meta-model (KDM) [15] and Abstract Syntax Tree Metamodeling (ASTM) [2] (specifications developed by the OMG ADM task force [1]), FAMIX [20], the Pattern and Abstract-level Description Language (PADL) [7], or the OASIS Static Analysis Results Interchange Format (SARIF) [19]. All these metamodels put their focus on artefacts such as source code, models, and specifications to extract knowledge from the software project. However, in addition to these artefacts, there are other less formal sources that are not usually considered and that can be processed and analyzed to get some extra knowledge about the software project being maintained or improved. These include forum discussions, issue tracker items, etc. In the PKM, we consider other less formal sources of knowledge that are poorly structured, incomplete, and sometimes incorrect. After a process of knowledge extraction, this information will be stored in the PKM.

To create and share knowledge throughout the different phases that are part of a software development process, different kinds of techniques to extract, process and store it are required. These techniques have been studied in different works in the literature. The extraction of meaningful data from software requirements is a tough problem that intends to formalise as best as possible knowledge that is informal and disparate. Several approaches have been developed with mitigated success. These approaches were possible thanks to recent research on innovative functions which range from de-obfuscating code [11], automatic bug fixing [8], natural language querying of API [17] comment generation of code [22] and even to code generation [16]. In this work we use an intermediate approach between manual knowledge elicitation and complete automation (as in the ARSENAL project [6]) since we consider that raw knowledge extracted from informal requirements is likely imprecise, incomplete and error prone, so the developer's intervention is needed. Therefore, this intermediate approach consists in extracting data automatically in a first step, and then, in allowing all manual corrections and refinements required to formalize the results progressively into valid formal specifications (e.g., ACSL/ACSL++/JML).

The problem of aggregating multiple types of knowledge within a project can be overcome with techniques for aggregating and disseminating knowledge, which are mostly based on GIT or SVN repositories. Nevertheless, these are inactive and do not provide much help to the developer when he/she requires knowledge about specific parts. Repositories do only manage source code, scripts and some documentation but no specifications nor models. In this work we based on the definition of a new Knowledge Base for handling (i.e., storing, retrieving, merging and checking for consistency) different kinds of knowledge and information related to a software project.

3. PKM Metamodel

Some software projects may produce artefacts that include informal data such as informal specifications, internal documentation, or even comments in the source code. The generation of formal documentation and code summarization provides useful information for users who have created that piece of code and have to return to it at some point as well as future maintainers. Filtering the content of informal data is a major hurdle, as this kind of data is poorly structured, incomplete, and sometimes incorrect. The extraction process is tedious as filtering needs some understanding of the language, its grammar, its semantics, and the context. For instance, documentation on GUI needs some understanding of widgets, buttons, call-backs, etc. Extracted information becomes useful when it can be formalized, either as models, specifications, or assertions on the code. This knowledge will be poured into the PKM for project stakeholders to use it whenever they deem it necessary during developments, for instance during corrective maintenance, in order to understand quickly what the current code does.

The PKM provides the representation of a general and specific knowledge about the artefacts of a software project. In order to manage the complexity of the PKM, it is defined

by a collection of metamodels according to the categories of the artefacts and a core package that defines the general knowledge of them. A first version of this PKM metamodel was introduced in [21]. The different metamodels are represented in different packages and includes:

- **Core package** that defines the core part of the PKM representing the software project, the SDM, the different artefacts of a project and its related concepts.
- **Abstract specification package** that defines the metamodel elements of the formal specification describing, by means of pre, post and invariants, the behaviour of an associated source code. This abstract specification can be automatically generated or manually written by means of annotations (e.g., in ACSL, ACSL++, JML, etc.).
- **Source code package** that defines the part of the metamodel that refers to the artefacts that list human-readable instructions written by a programmer with the objective of being executed in a computing device.
- **Report package** that defines the part of the metamodel that represents the artefacts containing a structured content in natural language, related to some particular chunk of source code or of a global nature.
- **Model package** that defines the part of the metamodel that represents abstract representations of a specific aspect from a given domain (e.g., a UML class model describes the structure – concepts, properties of the concepts, relationships between concepts- of a specific domain).
- **Configuration package** that defines the metamodel that represents artefacts describing, in plain text, the parameters that define or execute a specific software program.
- **Structured data package** that defines the metamodel that represents artefacts that store data structures and that are usually used as interchange format.
- **Extracted information package** that defines the metamodel that represents information produced by static source code analysis, by optimization passes of compilers, by natural language processing or by machine learning techniques.

Fig 1 shows the PKM Core Package. A software project is composed by different artefacts created by different tools throughout the development process. Artefacts are digital products or documents created in a development phase. Artefacts can be presented in different formats (plain text, key-value structures, markup documents), and levels of abstraction (high, medium, and low). Moreover, artefacts can be related to other artefacts with the same (or similar) semantic intention (e.g., a java file may be related to a UML diagram describing a class).

To deal with the heterogeneity of the data and knowledge stored in the PKM, this repository is implemented as a document-oriented database in MongoDB [10] where artefacts are transformed into JSON documents defined according to a specific JSON schema. A JSON Schema [12] specifies a JSON-based format to define the structure of JSON data for validation, documentation, and interaction control. The data used in the project is structured into complex documents, grouped into *collections*. Documents are linked together by means of internal fields such as file names, function names or identifiers. In this way, we provide a common language where the data and knowledge are structured into complex documents that are grouped into different collections (e.g., source code, annotations, documentation, tests, etc.). The complete JSON schemas are available in a gitlab repository[2].

In order to feed the PKM with the different artefacts of a project, we have developed a set of parsers for transforming the artefacts into JSON documents. For example, we have developed a Java Parser[3] to process all the java source code files contained in a Java project. Each java source code file is separately processed and translated into three different json documents: a first one describing the abstract syntactic structure of the source code (i.e., its AST), a second one describing, in a structured way, the comments included in the source code file, and a third one describing again, in a structured way, the annotations included in the source code file (e.g., JML annotations specifying preconditions, postconditions, and

[2] https://gitlab.ow2.org/decoder/pkm-api/-/tree/master/schemas/pkm-metamodels
[3] https://gitlab.ow2.org/decoder/pkm-api/-/tree/master/javaparser

invariants for the implemented source code). For C and C++ source code, Frama-C[4] and Frama-Clang[5] respectively have been extended to generate the three json files in the same way that our Java parser does for java code. Also, model-to-model transformations have been defined to support the translation between UML models into JSON documents.

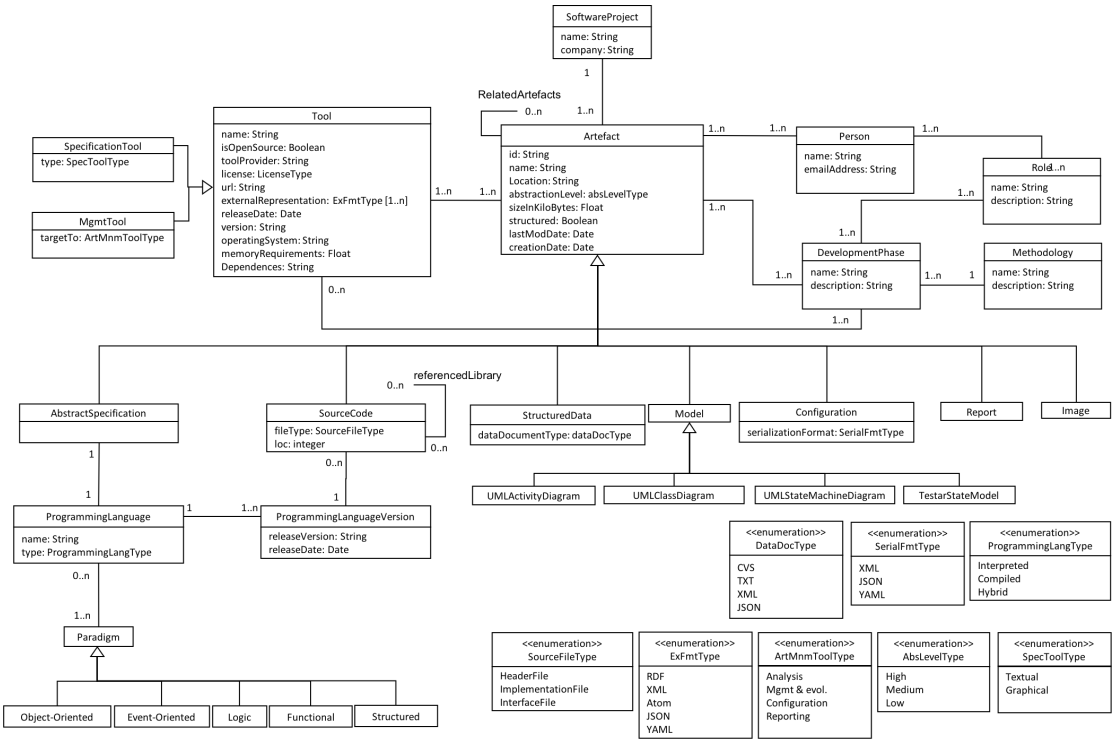

Fig 1. PKM Core Metamodel Package

Furthermore, the PKM keeps a traceability matrix that maps and traces the different artefacts stored in the PKM. This matrix is based on a tool that computes the similarity relationship between two given artefacts of a software project. In order to query the traceability matrix, a service has been defined that returns the degree of similarity of two artefacts of the same project. Finally, the PKM can be seen as an extension for Git repositories that provide semantic traceability between the stored project artefact. In this sense we plan to develop a service in charge of synchronizing the PKM content with the related git servers. To this end, the PKM will provide a representation of the tracked Git repositories that will be referenced by the stored JSON documents.

3.1. The PKM in the lifecycle of a software project

All the knowledge generated and gathered in the PKM will be used along the different stages of the software lifecycle to improve and assist stakeholders in their respective tasks. Fig 2 provides an overview over the different roles identified in DECODER and that can be involved in any software project as well as their interaction with the PKM. According to this figure, *developers* feed the PKM with the bulk code and documentation of the use cases where they are involved. Then, reviewers ask the PKM to generate partial source code formal annotations (in ACSL, ACSL++, or JML) from the documentation and the use-cases. These annotations contain invariants and behaviors implicitly connected to the artefacts that they derive from defining a new abstraction level (e.g., abstract state machines) of the code. However, these annotations are often incomplete to succeed a formal proof and *reviewers* have to correct and complete them before source code can be formally verified by means of tools such as Frama-C or openJML.

[4] https://gitlab.ow2.org/decoder/pkm-api/-/tree/master/frama-c
[5] https://gitlab.ow2.org/decoder/pkm-api/-/tree/master/frama-clang

Finally, based on the traceability matrix, *maintainers* prepare the next reviewing work and take decisions on how to resolve inconsistencies found in the different artefacts that conform the software project.

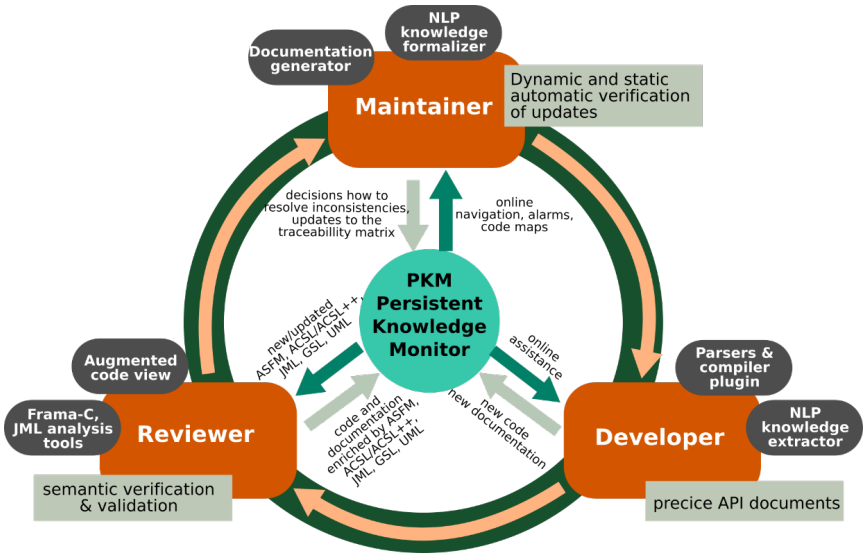

Fig 2. Interaction between the PKM and the stakeholders to generate/consume knowledge

4. Extracting knowledge from the PKM

This section introduces the way to extract data from the PKM by using a query language that can be useful for the three aforementioned roles participating in a software project. All of them can require information from the project that cannot be easily obtained.

Most software project repositories enable the retrieval of information based on syntactic information contained in data, just by typing the keyword or phrase into a query form. Nevertheless, we aim to not only find keywords but to find structural information contained in the different artefacts of the PKM and also concepts related to the software project and the methodology followed. For example, a software developer that has been recently involved in a project may require to discover which are the Java libraries used by other software developers or which is the code practice used for implementing loops. Therefore, the software developer is not interested in artefacts where the keyword 'library' appears or the term 'loop' appears; he/she is interested in searching for structural elements that are contained in some artefacts of the software project. The structure of the PKM allows software developers to make this kind of queries and obtain precise results. The proposed query language is oriented to facilitate the retrieval of this data.

4.1. Requirements of the query language

To define the query language, we first determine which may be the needs of the project stakeholders involved in a software project:

- Project stakeholders need to make queries, so, the language just **require a selection sentence**.
- The query is related to data that may be required by project stakeholders in a software project. We base on the PKM metamodel to determine which can be this data. The metaclasses, attributes and relationships of this metamodel are candidates to be retrieved as information. Also, as DECODER has defined a schema for the different artefacts that can be produced in a software project, (e.g., source code, documentation, UML models, comments, annotations, etc.), this enables the retrieval of specific elements of each artefact, such as elements of the AST for the source code, elements of the UML models, comments, etc. Taking into account the set of metamodels/schemas that make up the DECODER ecosystem, examples of data to be extracted from the PKM may be: libraries used in the project, tools that have been applied, development phases that have been followed, roles participating

in the project, people involved in the project, different kind of artefacts that have been created, structures of the source code (e.g., loops, conditions, comments), structures of different diagrams (e.g., derived classes in a class diagram or conditions in a sequence diagram). Therefore, the selection request sentence requires **a clause to determine the elements of the metamodel from which instances have to be searched and retrieved**.

- Project stakeholders could be interested in bounding the searching to the instances that fulfill a specific condition. For example, a participant could be interested in programming languages that belong to a specific paradigm, structured data that conforms a specific format, or roles that play a specific person. Therefore, the language requires a **clause to determine a condition that must fulfil the instances to be selected and retrieved**.
- The PKM stores elements grouped in collections that have been defined according to the different artefacts in a software project. This leads to different sources in the database. The existing collections are the following: *PKMCore* (all the information related to the PKM core metamodel), *SourceCode* (all pre-processed source code (C, C++ and Java)), *RawSourceCode* (the unprocessed files with the source code, whose processed version is in the previous collection), *Annotations* (the annotations of the Sourcecode), *Comments* (all the informal comments attached to the source code (list of strings)), *ClassDiagram, StateMachineModels*, *UseCaseDiagram*, *Documentation* (all the informal documentation associated to the project), *Logs* (all the log files of the project, produced along different activities on the other collections), *TraceabilityMatrix* (hyper-links between documents in the other collections, to implement the traceability matrix), *TestResults (*the results of testing tools). The data to be selected and retrieved could be bounded to a specific source. Therefore, the query language requires a **clause for specifying from wherein the element should be extracted**.
- The query **should return instances of the elements of the metamodel** satisfying the specified condition.

4.2. Query language

In order to make queries in the PKM, we used the MongoDB query language [13]. MongoDB provides the *find* function to retrieve documents from a MongoDB database.

Nevertheless, the *find* sentences are strongly dependent on the structure of the JSON files, which should be transparent for users of our query language. To allow technological independent queries, we wrapped the MongoDB query language with a high-level language that allows making queries that are near to the users' needs. This language splits a selection sentence into three main parts, which refer to the scope of the query (Source), to the type of element being retrieved by the query (Concept), and the condition to be satisfied by the retrieved instances (Condition). To illustrate these parts, we make use of the following scenario: let us consider a software development project to manage bank customers and their accounts. Consider now that a developer modifies the attribute *accountNum* of the *BankAccount* class. After this modification, the developer wants to make sure that this change does not introduce any inconsistency with the artefacts related to it, e.g., requirements, documentation, UML models (class and use case), etc. To this end, the developer may query the PKM to identify which are these related artefacts and which parts need to be checked. Let us consider that the developer wants to query the operations specified in the UML class diagrams that are affected by the modification of such attribute. The query to be constructed in this case would be the following:

```
1. CONCEPT operations
2. SOURCE ClassDiagram
3.          RELATED TO compilationUnit
4.          IS "accountNum"
5.          SOURCE SourceCode
```

Next, we use this query to explain in more detail the three main parts of the query language and to show how to retrieve the interesting knowledge from the PKM.

Source. It determines the PKM collections from where to do the search, i.e., it delimits the scope of the search to perform. If this value is not specified, it means that the user wants to query all the collections of the PKM. The selection sentence is constructed with the keyword SOURCE followed by the name of the collection. The collections that can be used to narrow the query are the ones presented in Section 4.1. Thus, the source part of the query in order to search on the ClassDiagram collection is the one showed in line 2.

Concept to be searched. It determines the types of elements that are going to be retrieved by the query from the collection determined in the SOURCE part. It can refer to any element that is part of the set of metamodels that make up the DECODER ecosystem, i.e., it can refer to a metaclass (e.g., SourceCode, ForStatement, Tool, Person, Loop, etc.), to an attribute of a metaclass (e.g., the filetype attribute of the SourceCode metaclass), or to a relationship between metaclasses (e.g., the referencedLibrary reflexive relationship of the SourceCode metaclass). The selection sentence is constructed with the keyword CONCEPT to determine the elements to be searched. Following with the scenario, to retrieve the methods defined in the class diagrams stored in the ClassDiagram collection, we may add to the SOURCE statement defined previously, the CONCEPT statement specifying the metaclass "operations" as shown in line 1.

Condition that has to be fulfilled. It determines the condition that has to be fulfilled by the retrieved instances. This condition can be defined from the non-explicit relationships that exist between different artefacts (e.g., source code and a UML class model) due to the similarity of their content and that can be extracted from the traceability matrix (*Condition from non-explicit relationships*) or from the explicit relationships that are defined in the metamodel (*Condition from explicit relationships*).

- *Condition from non-explicit relationships*: This type of condition will be used when we are interested in retrieving an artefact whose content is highly similar to the content of another one. This condition is defined as a nested query (see lines 3-5) where we should specify the related artefact by using the RELATED TO keyword (see line 3). This kind of condition is possible since the PKM keeps, in a traceability matrix, links and cross-references between the artefacts of the PKM.
- *Condition from explicit relationships*: This type of condition will be used when we are interested in retrieving elements from the metamodel that have (or are related to) an element (metaclass, attribute or relationship) whose value matches with the value specified in the condition. This condition can be applied in both queries, i.e., the main one and the nested one. In both cases, this element can be defined by the own element specified in the CONCEPT/RELATED TO part of the corresponding query or by the elements that are reachable through the existing relationships. Besides, we can specify whether we are looking for an exact match (by using the IS keyword) or for a partial match (by using the AS keyword) and then indicating the literal to be searched. According to the proposed scenario, and as shown in line 4, we should use the IS keyword since we know the exact literal we are looking for. However, if this would not be the case, we could also use the AS keyword followed by, for example, the "account" literal. In this case, this literal would be used as a substring in the final search, retrieving as a result all the related elements that contain such substring (e.g., lockAccount, numOfAccounts, etc.).

As a result, the constructed query would return the class operations that have a relationship with any source code that either contains the literal "accountNum" or is related with any element that contains the literal "accountNum".

To actually query the PKM we should translate this query into a query in the MongoDB query language. This MongoDB query sentence performs a search over the PKM. The translation is made based on several templates shown in Table 1.

Table 1. Templates used to transform queries to MongoDB query

1	`CONCEPT` **`concept`**`,` `SOURCE` **`source`**	`db.getCollection('source').find({},{concept:1})` If the source is not specified, a query for each collection is generated.

2	`CONCEPT` **`concept`**
	`IS` **`"feature"`**
	`SOURCE` **`source`**
		`db.getCollection('source').find({$or:`
		`[{concept.field1:"feature"},`
		`{concept.field2:"feature"}…]},{concept:1})`

		Where field1, field2 are the different fields of the specified concept and its related concepts.
3	CONCEPT **concept** AS **"feature"** SOURCE **source**	db.getCollection('source').find({$or: [{concept.field1:{$regex:/feature/}}, {concept.field2:{$regex:/feature/}}…]},{concept:1}) Where field1, field2 are the different fields of the specified concept and its related concepts.
4	CONCEPT **concept** SOURCE **source** RELATED TO **concept2** AS **"feature"** SOURCE **source2**	A first query obtains the id of the artefacts associated to concept2: db.getCollection('source2').find({$or: [{concept2.field1:{$regex: /feature/}}, {concept2.field2:{$regex: /feature/}}…]},{_id:1}) A second query obtains all the ids associated to the artefacts of the collection 'source': db.getCollection('source').find({},{_id:1}) Then, the Traceability Matrix service is invoked to obtain the degree of relationship between the artefacts: curl -X POST https:// ow2- decoder.xsalto.net/decoder/traceability/ -h {key:LOGIN_KEY} -d {artifact1:ARTEFACT_1_ID, artifact2:ARTEFACT_2_ID} This service returns a JSON object that specifies the degree of relationship between two artefacts with the following format: {artifact1:ARTEFACT_ID,artifact2:ARTEFACT_ID, relationshipDegree:REL_DEGREE} Finally, we select only those artefacts with a high degree of relationship and performs the following query: db.getCollection('source').find({$or:[{_id:ARTEFACT _ID},{_id:ARTEFACT_ID}...]},{concept:1}) where ARTEFACT_ID are the different IDs of the artefacts with a higher degree of relationship

5. Case study evaluation

This case study presents an evaluation to analyse the usability of the query language to extract knowledge from the data stored in the PKM. To do so, we arranged an experiment in which participants played the role of software developers involved in a Java project which is stored in the PKM. To achieve this, we populated the PKM with data from the Java project so participants can query it to extract knowledge out of it. The population task was performed using the set of tools developed in DECODER to process and store different types of artefacts (e.g., source code, UML models, documents, etc.) as JSON documents according to the schemas defined in the DECODER project.

The case study evaluation was conducted following the research methodology practices provided by Runeson and Höst [18], which describe how to conduct and report case studies and recommend how to design and plan the case studies before performing them. In this section we summarize the experiment and the results obtained, but the complete description of the experiment can be found in a technical report[6].

5.1. Design of the case study

The provided case study is a Java project whose main goal is to manage customers and their accounts (a.k.a Banking App). Within this context, participants are asked to join the Banking App project team to include some new functionality and also to verify the Java source code developed in the project. However, prior to this task, and to have a better understanding over the Banking project, participants are asked to query the PKM to extract knowledge related to the given task.

A total of 15 subjects between 24 and 39 years old participated in the experiment (four female and eleven male). They were students of the Master's Degree in Informatics Engineering and were recruited through personal invitation.

In order to analyse the usability, we based on to the standard ISO 9241-11 (1999) which states that the main affected variables concerning usability requirements are (1) effectiveness and (2) user acceptance. While the effectiveness was measured as the grade of task completion reached by the participants compared with a predefined master result,

[6] http://tatami.webs.upv.es/decoder/isd/CaseStudy.pdf

user acceptance was measured by means of a TAM questionnaire. Thus, the instruments that were used to carry out the experiment were:

- A demographic questionnaire.
- A task description that introduced the queries that the participants had to carry out during the experiment[7].
- A TAM questionnaire to evaluate the perceived usefulness and perceived ease of use of the proposed query language.
- A rubric to evaluate the effectiveness of the query language by means of different grades according to the task completion.

5.2. Execution of the case study

To perform the experiment, we arranged a one-day workshop with two sessions of three and four hours long respectively. In the first session, participants were proposed to fill in a demographic questionnaire to capture their background and were trained in our query language and also the PKM. In the second session, participants were introduced to the banking java project. From this starting point, participants played the role of developers and have to add functionality for enabling the modification of the customer information. In order to achieve this, the participants were asked to define, using the query language, four queries that extract knowledge that could help them to add this new functionality. The queries were the following (we show here the solution of the query to exemplify the query language):

- Q1: Whether the already developed source code was verified by someone else, and if so, the results obtained after this verification.

```
CONCEPT openJMLReport
SOURCE TestResults
       RELATED TO compilationUnit
       AS "BankAccount"
       SOURCE sourceCode
```

- Q2: Whether there is any tool available to validate the implemented java source code. And if so, which one?

```
CONCEPT Tool
SOURCE PKMCore
       RELATED TO DevelopmentPhase
       AS "validation"
       SOURCE PKMCore
```

- Q3: Which UML class diagram is affected when I implement this new functionality in the Java source code?

```
CONCEPT UMLClassDiagram
SOURCE ClassDiagram
       RELATED TO compilationUnit
       AS "BankAccount"
       SOURCE SourceCode
```

- Q4: Who was in charge of defining/reviewing the JML specifications of the already developed Java source code?

```
CONCEPT Person
FEATURED "reviewer"
SOURCE PKMCore
       RELATED TO compilationUnit
       AS "BankAccount"
       SOURCE SourceCode
```

After performing the queries, each participant had to fill in the TAM questionnaire. Throughout this session, we observed participants and took notes on their behaviour. After the task, we filled in the rubric to evaluate effectiveness for each participant and query.

[7] The task description document can be downloaded from: http://tatami.webs.upv.es/decoder/isd/taskDescription.pdf

5.3. Analysis of results

Next, we present and analyse the results obtained from the above-introduced experiment regarding effectiveness and user acceptance.

Effectiveness. We measured the effectiveness as the grade of task completion in such a way a query was complete and correct. The grades of the rubric were: suitable query (10 points), suitable but not complete query (7 points), not appropriate query (4 points), or not capable to build the query (0 points). To facilitate this evaluation a master query was used as a reference point. After performing the evaluation, we obtained an average mark of 6.72 over 10. Although this note indicates that the language can still be improved, it also indicates that the proposed query language is effective enough to extract the knowledge that is required at some point of the development process. Note also that all the participants had some experience in JSON and SQL, and this had a negative impact on the use of the proposed query language. Specifically, we detected the following problems:

- Participants had problems to define conditions due to their background in SQL. Specifically, they tried to build the condition statement similarly to they would do it in a SQL WHERE clause.
- Participants had difficulties to specify the CONCEPT statement at the beginning because they were not familiar with the JSON schemas used in the PKM.
- Participants had troubles to decide when to use non-explicit or explicit condition.
- Participants had difficulties in understanding how to specify the conditions in the non-explicit relationships because they tend to compare it with the JOIN operator in SQL.

User Acceptance. According to the TAM results, the designed task was ranked with values that illustrate that participants perceived the query language as not so easy to use as expected. Users thought that with the query language they do not complete queries more quickly than with other query languages but it could enhance their productivity, making their job easier. For that reason, they found the query language to be useful in their job.

In general, they did not find the query language easy to learn. This was reflected during the experiment, where participants asked a lot of questions about the language. Also, they rated the query language as not much flexible to interact with. However, they found easy to become skilful when they use the query language because once learned, it is simple to use. Finally, participants mentioned that the use of the query language could be easier if they have had a supporting and assistance tool for query construction.

6. Conclusions

In this paper, we have introduced the PKM metamodel, a metamodel designed to represent knowledge from the different artefacts that are part of a software system/project. Such metamodel allows detailing for each type of artefact contained in a software project its content, the tools used to edit and manage it, the persons involved in the artefact management process, the stages in which these participate within a specific development process or methodology and finally the history of changes which artefacts have been exposed to along their life. The metamodel has been designed in blocks or packages to make it extensible whenever it is required. This metamodel, once populated, is used to derive new knowledge that can be used and exploited to better understand the different artefacts (source code, documentation, models, etc.) that form a software project. Moreover, this knowledge is useful for the different actors involved during the life span of a software, especially new persons, to keep project information as accessible and unambiguous as possible. This living repository can be queried and enriched by the actors involved in the project, in order to maintain consistency and keep the most updated and precise information about it.

By having all the knowledge of the project structured in the PKM, project stakeholders can make queries that aim not only to find keywords but to find structural elements that appear in the components of the artefacts that are part of a software project. We have introduced a query language that allows making queries with an enclosed syntax. This is just the first approach to a language for extracting knowledge of the PKM. Further work will be dedicated to extending the language to make more advanced queries and to validate

it in different software projects. Finally, we aim to develop a GUI to assist and guide project stakeholders in constructing the queries.

Acknowledgements

This work has been developed with the financial support of the European Union's Horizon 2020 research and innovation programme under grant agreement No. 824231 and the Spanish State Research Agency under the project TIN2017-84094-R and co-financed with ERDF.

References

1. ADM initiative website. http://adm.omg.org. Accessed 5 July 2019
2. Architecture-Driven Modernization: Abstract Syntax Tree Metamodel (ASTM), OMG document formal/2011-01-05, OMG, Jan. 2011. [Online]. Available: http://www.omg.org/spec/ASTM
3. Chau, T., Maurer, F., Melnik, G.: Knowledge sharing: agile methods vs. Tayloristic methods. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003, pp. 302-307.
4. Cockburn, A., Highsmith, J.: Agile Software Development: The People Factor. IEEE Computer, 34(11), 131-133 (2001)
5. Ebert, C., Gallardo, G., Hernantes, J., Serrano, N.: DevOps. IEEE Software, 33(3), 94-100 (2016)
6. Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., Steiner, W.: ARSENAL: Automatic Requirements Specification Extraction from Natural Language. In: arXiv:1403.3142 [cs.CL] (2016).
7. Guéhéneuc, Y.G.: "Ptidej: promoting patterns with patterns", in 1st ECOOP Workshop on Building Systems using Patterns, pp. 1-9. Springer, Heidelberg (2005)
8. Home | Pliny: Big Code Analytics (2018), http://pliny.rice.edu/. Accessed March 10, 2021
9. Ibraheem Y.Y. Ahmaro, Abdallah M. Abualkishik and Mohd Zaliman Mohd Yusoff, 2014. Taxonomy, Definition, Approaches, Benefits, Reusability Levels, Factors and Adaption of Software Reusability: A Review of the Research Literature. Journal of Applied Sciences, 14: 2396-2421.
10. Inc MongoDB. 2014. Mongodb. URL https://www.mongodb.com/. Accessed April 01, 2021.
11. JS NICE: Statistical renaming, Type inference and Deobfuscation (2018), http://jsnice.org/. Accessed March 10, 2021
12. JSON Schema and Hyper-Schema, json-schema.org. Accessed February 15, 2021.
13. MongoDB CRUD Operations, https://docs.mongodb.com/manual/tutorial/query-documents/. Accessed March 10, 2021
14. Nembhard, Fitzroy & Carvalho, Marco & Eskridge, Thomas. (2018). Extracting Knowledge from Open Source Projects to Improve Program Security. 1-7.
15. Object Management Group, Inc. (2012) Knowledge Discovery Meta-model (KDM). [Online]. Available: http://www.omg.org/technology/ kdm/index.htm
16. Pengcheng, Y., Graham, N.: A Syntactic Neural Model for General-Purpose Code Generation. arXiv:1704.01696 (2017)
17. Richardson, K., Kuhn, J.: Function Assistant: A Tool for NL Querying of APIs, Proceedings of EMNLP 2017, Jun. 2017
18. Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. Empirical software engineering, 14(2), 131–164
19. Static Analysis Results Interchange Format (SARIF) website. https://www.oasis-open.org/committees/sarif. Accessed March 10, 2021
20. Tichelaar, S., Ducasse, S., Demeyer, S.: "FAMIX and XMI," Proceedings Seventh Working Conference on Reverse Engineering, Brisbane, Queensland, Australia, 2000, pp. 296-298. doi: 10.1109/WCRE.2000.891485
21. Torres V., Gil M., Pelechano V. (2019) Software Knowledge Representation to Understand Software Systems. In: PROFES 2019, pp: 137-144, vol 11915. Springer, Cham.
22. Wang, X., Yifan, P.: Comment Generation for Source Code: State of the Art, Challenges and Opportunities. arXiv:1802.02971 (2018)