MEntoR: Mining Entities to Rules[☆]

Angela Lozano, Andy Kellens, Kim Mens, Gabriela Arevalo

Université catholique de Louvain, Vrije Universiteit Brussels,Université catholique de Louvain, Universidad Austral

Abstract

MEntoR is a source code analysis tool that is integrated into the development environment (IDE) to suggest implementation improvements. MEntoR is a front-end for Prospector. Prospector extracts properties of source code entities of an application, and calculates association rules which indicate which properties tend to occur together. The purpose of MEntoR is to identify missing properties in the source code entity that is being browsed in the IDE based on other properties that the entity has. For instance, showing whenever a source code entity violates an idiom.

Keywords: Source code mining, IDE support, implementation suggestions

1. Introduction

When maintaining or extending an existing software system, developers frequently have to interact with source code which they are unfamiliar with. Consequently, a considerable amount of time is spent on understanding the source code, before actual changes can be made to the system. To adapt a system, the developer needs to be aware of the various structural regularities such as naming conventions, implementation idioms, design patterns, architectural dependencies, and so on that govern the source code entity that is about to be changed. Documentation of these regularities however is often lacking or outdated. While such knowledge might be available from the original developers, access to these developers might be limited. As a result, the process of identifying which structural regularities need to be followed by a source code entity remains a largely manual effort.

In previous work [1] we have reported on a semi-automatic approach for mining structural regularities from source code (Prospector). Such approach is based on the assumption that, even when these regularities are not explicitly documented, they can be retrieved from the source code. By using the technique of association rule mining, we have been able to successfully extract naming conventions, API definitions, and co-dependent methods from class definitions.

In this paper we present a coding assistant tool — named MEntoR— that leverages the information obtained by our mining algorithm to provide developers suggestions regarding related source-code entities and applicable regularities, based on the current browsing context. Whenever a developer is browsing a particular source-code entity, MEntoR presents a detailed overview of all regularities (mined using Prospector) that are related to the entity, and that are of potential interest to the developer. For each presented regularity, the tool indicates whether the browsed entity respects this regularity or not. To ease interpretation of the regularities, our tool provides access to all other source-code entities respecting the regularity, as well as to all violations of the viewed regularity. Other features of our tool include a tagging mechanism that allows developer to identify and document potentially interesting regularities, and a visualization that displays a regularity in the context of the entire system.

[☆]This work has been performed in the context of the MinDeR bilateral project sponsored by MINCyT Argentina and FWO Flanders. Angela Lozano is funded by the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB). Andy Kellens is funded by a research mandate provided by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen). This work has also been supported by the Interuniversity Attraction Poles (IAP) Programme of the Belgian State – Belgian Science Policy.

2. How Prospector's mining work

This section explains Prospector, which is the back end for MEntoR. The aim of Prospector is to help users in rapidly inferring and documenting the relevance and rationale of potential regularities. Examples of regularities found by our approach include naming conventions, idioms, implementation protocols, design pattern constraints and implementation improvements. Prospector is based on a generic lightweight association rule mining. Its input is a set of source code entities (classes, methods) along with properties of those entities (inherit from, implements, contains in its name, calls, etc.). Its output is a set of association rules that express interesting source code regularities. Association rules indicate that a discriminative set of properties are commonly co-occur with another discriminative set of properties. Association rules have the form of a fuzzy conditional clause, like, *"if an entity exhibits properties* `x`*,* `y`*, and* `z` *then it has a likelihood* `c` *of also having properties* `r` *and* `s`*"*. A more specific example would be: *"if a class* `belongs to the hierarchy of Action`*, then it always (*`100% likelihood)` `implements the method performAction`"*[1] which indicates the implementation of the command pattern. The first set of discriminative properties (e.g. `x`, `y`, and `z` ; or, `belongs to the hierarchy of Action`) is called the condition of the rule. The second set of discriminative properties (e.g. `r` and `s`; or, `implements the method performAction`) is called the conclusion of the rule.

Our mining approach consists of the following steps:

1. Determine the kind of input we give to the mining tool. In our previous work [1], we restricted our analysis to classes only, to minimize the amount of information to analyze. After several tests, the properties selected for classes were the *hierarchy* (superclasses of the class), the *implement* relations (names of the methods implemented by the class), and the *identifiers* (words contained in the name of the class).
   We modified Prospector, so that MEntoR could present method properties. For each analyzed method we extract the following properties: *identifiers* (words contained in the name of the method), *belongs to* (the name of the class implementing the method), *protocol* (the name of the Smalltalk protocol in which the method classified), *calls* (the signature of the methods called by the method), *super-calls* (the signature of the methods of the super class called by the method), and *refers* (the names of the classes mentioned in the implementation of the method).
2. Pre-filter input data to improve the performance of the mining technique the tool prunes redundant and trivial information. Before the executing the algorithm, we remove properties that are redundant or irrelevant to our analysis. For instance, omnipresent classes or common words generate trivial rules such as all classes are in the hierarchy of Object, or all methods named *get*.
3. Apply an association rule mining algorithm in order to obtain a set of implication and association rules. We calculate association and implication rules using an efficient Formal Concept Analysis (FCA) algorithm (more details can be found in [1]).
4. Post-filter (simplify and prune) resulting rules to obtain the most concise and expressive set of rules.Structural filters are based on structural properties of the mined association rules and aim both at eliminating irrelevant rules (rules subsumed by other rules), and making the rules more concise. For example, prune rules with confidence less than 70%[2]. Intuitively, if a rule is valid for less than 70% of the elements satisfying the condition of the rule, we do not consider it to be an interesting regularity because there is too much counter-evidence. Heuristic filters use domain knowledge to further restrict the amount of (meaningless) results. They rely on the semantics of the different kinds of analyzed properties in order to further restrict the set of mined association rules. For example, regarding hierarchy relations our algorithm identifies implication rules that express mere subclass relationships (e.g., if a class is in the hierarchy of class A, it also is in the hierarchy of class B which is trivially true for each superclass B of A). Since such implication rules present trivial information, they are pruned from the final result. After post-filtering an application of approx. 300 classes remains with 100 to 150 rules.

[1] The example would be shown in the tools as `Hier>Action` ⇒ `Impl>performAction`. Note that each type of property has an abbreviation e.g., Hier> for *hierarchy*, Impl> for *implement*, Cid> for class *identifiers*, etc. The arrow indicates the direction and likelihood of the rule. The arrow also indicates if the rule has condition exceptions (→), or not (⇒).

[2] All thresholds used to filter rules are determined per case study. The case studies analyzed so far have had similar thresholds, we hypothesize that ideal thresholds depend on the size of the application.

5. Group sets of related association rules to convey the mined results in a more concise manner. The number of association rules can be considerably large. However, in many cases, several association rules describe properties of the same set of source code entities, and are often contributing to the same regularity. We consider two association rules to belong to the same group (i.e., contributing to the same structural regularity) if at least 70% of their matches overlap.

Overall, the mining approach of Prospector (in which MEntoR relies) has the following properties:

- Intensional representation: We provide the user with enough information to allow him or her to verify, infer or codify the underlying intent, instead of just providing a mere enumeration of entities that comply with a predefined regularity.
- Robustness towards deviations: Since association rules behave like fuzzy conditional clauses, the technique is robust towards deviations, and can thus detect regularities even in the presence of deviations.
- Conciseness of results: Our filters reduce the total amount of information presented to the user and increase the quality of that information.

Prospector finds regularities, i.e. groups of properties shared by several source code entities. Nevertheless, given that the inspection of regularities must be done manually MEntoR proposes an alternative way to present results: in context and by demand. That is, by showing only the association rules that are relevant to the source code entity that is being browsed.

3. MEntoR's features

MEntoR is a tool designed to enrich the way in which a source code entity is presented to a developer so that (s)he can infer the concepts and features related to the entity. MEntoR is seamlessly integrated with the IDE in order to provide timely and non-intrusive information of the source code entity in focus. Given that Prospector is developed in Smalltalk, we decided to integrate MEntoR as a plugin for VisualWorks[3]. Although the Prospector can handle source code written in Java and Smalltalk, MEntoR is only available for VisualWorks and therefore it can only handle Smalltalk code.

The purpose of MEntoR is to provide contextual information for novice and expert developers. Novice developers benefit by tracing related source code entities that provide examples, by gathering rationale about the application's design by looking at diverse properties that are key to the source code entity in focus, and by browsing the rules to confirm (or reject) their hypotheses about how the application works.

Expert developers benefit from the identification of outliers to the rules which may signal degradation of implementation conventions. Such degradation can interfere with future maintenance as they conceal application knowledge whose only reliable location tends to be the source code. In the worst case such degradation may indicate implementation errors. Moreover, MEntoR can give implementation suggestions that can be useful for both novice and expert programmers.

MEntoR's layout (see Fig. 1) is divided in two areas: the left area shows the rules that are related to the selected source code entity(see area number 1 on Fig. 1), while the right area shows details of the rule currently chosen (with a click). The left area is divided into bottom, middle, and top area. The bottom left area presents association rules related to the source code entity in focus.

The association rules are divided into three lists. The first list indicates possible errors in the implementation of the source code entity in focus (see area number 3 on Fig. 1). Possible errors are detected as association rules in which the source code entity complies perfectly to the conditions of the rule but it does not comply with the conclusions. The second list indicates suggestions for the source code entity in focus (see area number 4 on Fig. 1). A suggestion is an association rule that is partially satisfied (i.e. has properties mentioned in the condition and conclusion of the association rule) by the source code entity. The third list indicates rules that are fully satisfied by the source code entity (see area number 5 on Fig. 1). Given that the first and second list refer to association rules partially satisfied by

[3] VisualWorks is a commercial IDE for Smalltalk that offers a free academic version.

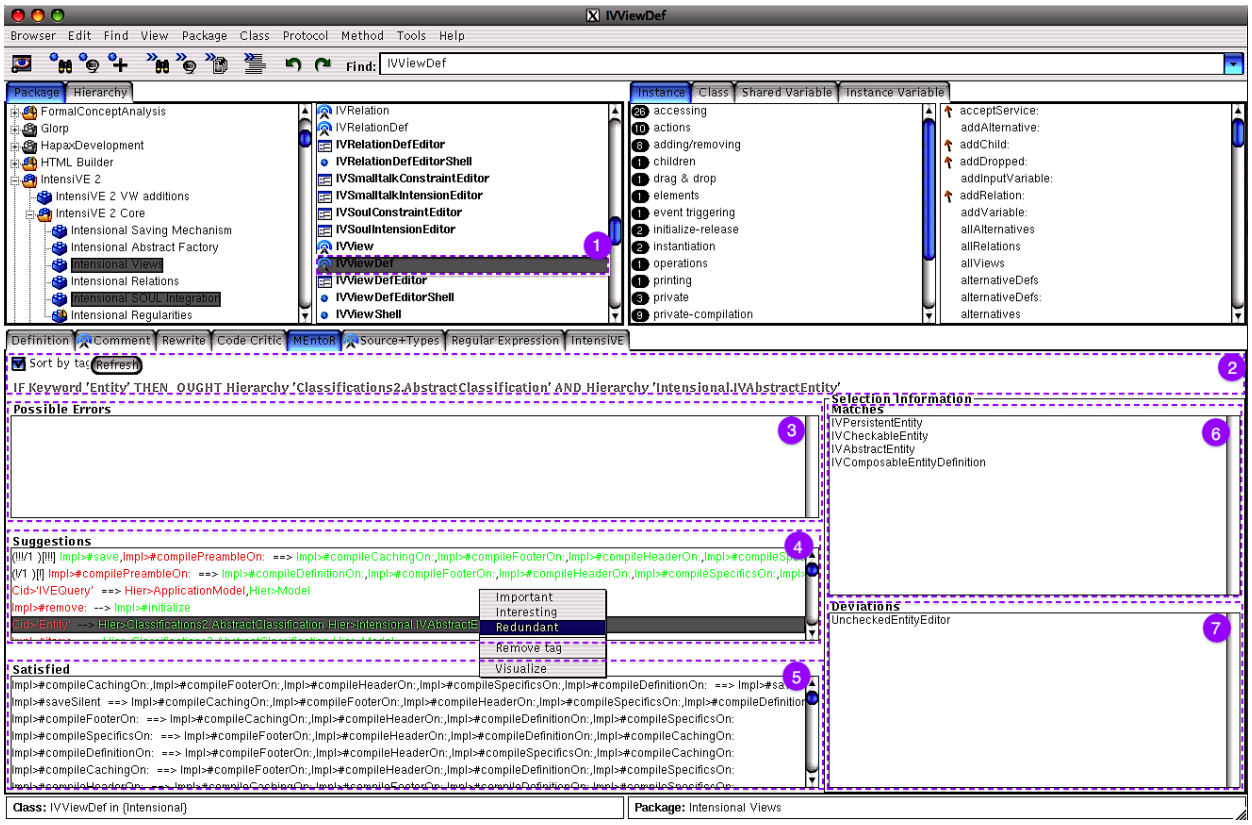

Figure 1: MEntoR's layout

the source code entity in focus, we give a visual hint to identify the characteristics present and missing in the source code entity with respect to the rule. Missing characteristics to comply with the rule are written in a red font, while present characteristics of the source code entity and the association rule are written in a green font.

Finally, the top of the left area (see area number 2 on Fig. 1) has a label, a check-box and a button. The label shows an easy-to-read version of the rule currently selected (from those available in the association-rules area i.e. areas 3, 4, and 5 on Fig. 1). The check-box allows the developer to sort all the association rules shown (i.e. related to the source code entity in focus) by tag importance. Those tagged as important appear at the top of each list of association rules; next, all rules tagged as interesting will appear; followed by, rules that have not been tagged yet; and finally, those rules tagged as redundant. Once the sorting by tag is checked, each rule will have its tag representation (*!!!* for important, *!!* for interesting, and — for redundant). All tagged rules indicate the number of votes and most voted tag e.g. *(!!!!3)* indicates that the the rule has been tagged three times, and that the most frequent tag for this rule is important. The refresh button allows the user to recalculate all regularities of the application of the source code entity in focus. This is a necessary step given that calculating regularities is computationally intensive because any change can modify the whole FCA-lattice, and therefore all the association rules.

The area on the right shows other source code entities related to the association rule currently selected in the area on the left. The area on the right is divided into two lists. The list on top (see area number 6 on Fig. 1) shows the matches of the rule, that is the source code entities that comply with the condition and conclusion of the rule currently selected. The list on the bottom (see area number 7 on Fig. 1) shows the deviations of the rule, that is the source code entities that comply only with the condition of the rule currently selected. By comparing the source code entity in focus with the matches and the deviations, the developer has a hint of whether or not the source code entity is similar to the entities described by the selected rule, and therefore if it should be changed to comply with the rule.

It might be that the source code entity is a deliberate exception to the rule, in which case it is useful to document that the rule does not apply to that source code entity (redundant or irrelevant). Once a rule is chosen it is also possible (right click menu) to document its usefulness with respect to the current entity, and to highlight the source code entities

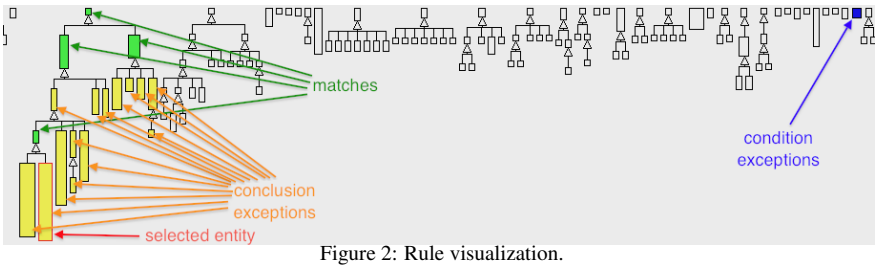

Figure 2: Rule visualization.

involved in a rule in a UML-like diagram (see Fig. 2).

4. Related work

MEntoR is a code suggestion tool based on the assumption that the source code entity browsed conveys the context of the development task. However validating the usefulness and pertinence of the information presented with respect to other code suggestion tools remains future work. Other code suggestion tools that aim at integrating the implementation contexts can be found in [2] and[3] Kersten and Murphy present *Mylar* (now Mynlyn) [2], a tool that offers a degree-of-interest model to capture the task context of a developer. Mylar associates an interest value with each source-code entity. Whenever a developer browses an entity, that entity's value is increased. Over time, an entity's interest value decreases if that entity is no longer browsed. Mylar aids a developer in capturing the task context by focussing on source-code entities with a high interest value. Mylar is largely complementary to our tool: while Mylar focusses on constructing the current task context, MEntoR aims at presenting the developer with related source-code entities and regularities. Robillard introduces a technique for proposing and ranking related source-code entities based on analysis of the topological structure of a program [3]. As input this technique takes a set of source-code entities. Based on the relations between these entities, and other entities in the system, it presents the user a fuzzy set containing related source-code entities. In contrast to our tool, this approach does not provide an intentional description of why entities are related (i.e., the mined regularity).

5. Future work

Currently we are designing a series of experiments to validate to what extent MEntoR improves the development process, and to confirm that the source code entity in focus permit an easier analysis of the regularities than in batch as Prospector does.

Besides MEntoR's usage is being improved by adding the following tags automatically:

- 'looked at' whenever the developer has clicked on a rule,
- 'confirmed' whenever a rule has been 'looked at' and its status has improved (from suggestions to satisfied, or, from possible errors to suggestions or satisfied),
- and 'ignored' whenever a rule has been 'looked at' and its status has degraded (from satisfied to suggestions or possible errors, or, from suggestions to possible errors).

We also plan to add a text box to allow the developer to store his/her rationale for the current rule. In this way the usefulness of the rules remains stored and can be consulted by developers and researchers. Finally, we would like to present the rules as overlapping Venn diagrams.

References

[1] A. Lozano, A. Kellens, K. Mens, G. Arevalo, Mining source code for structural regularities, in: Working Conference on Reverse Engineering, 2010, pp. 22–31.

[2] M. Kersten, G. Murphy, Mylar: a degree-of-interest moel for IDEs, in: International Conference on Aspect-Oriented Software Development, 2005, pp. 159–168.

[3] M. Robillard, Automatic generation of suggestions for program investigation, in: European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE), 2005, pp. 11–20.