Mendel: Source Code Recommendation based on a Genetic Metaphor

Angela Lozano
Université catholique de Louvain
Place Sainte Barbe 2
B-1348 - Louvain-La-Neuve, Belgium
Email: angela.lozano@uclouvain.be

Andy Kellens
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 - Brussels, Belgium
Email: akellens@vub.ac.be

Kim Mens
Université catholique de Louvain
Place Sainte Barbe 2
B-1348 - Louvain-La-Neuve, Belgium
Email: kim.mens@uclouvain.be

Abstract—When evolving or maintaining software systems, developers spend a considerable amount of time understanding existing source code. To successfully implement new or alter existing behavior, developers need to answer questions such as: “Which types and methods can I use to solve this task?”, “Should my implementation follow particular naming or structural conventions?”, “How is similar behavior implemented in the system?”. In this paper we present Mendel, a source code recommendation tool that aids developers in answering such questions. Based on the entity the developer currently browses, the tool employs a genetics-inspired metaphor to analyze source-code entities related to the current working context and provides its user with a number of recommended properties (naming conventions, used types, invoked messages, etc.) that the source code entity currently being worked on should exhibit. To validate our approach, we analyze to which extent Mendel is able to provide meaningful recommendations, by comparing our recommendations with the actual implementation of five open-source systems. The results seem to confirm the potential of our approach.

I. INTRODUCTION

Source-code regularities such as naming conventions and programming idioms [1] [2], code templates and patterns [3], play an important part in increasing a program’s comprehensibility. Developers introduce such regularities in the source code to make particular implementation and design concepts explicit. For example, when implementing a class hierarchy that represents various kinds of user interface actions, it is not uncommon to suffix the names of all classes in this hierarchy with ‘Action’. Similarly, methods related to the same task are often not only related in name, but may also share other traits. For example, they may have the same structural pattern, use the same types, or invoke the same methods.

Developers who need to extend or maintain a piece of source code often spend a considerable amount of time understanding this source code [4]. To successfully make their changes, they need to be aware of the various regularities that govern the piece of source code that is being changed. As these regularities [5] are often not explicitly documented, this can be a non-trivial task.

In this paper we present a novel code assistant algorithm for object-oriented systems named Mendel1, along with a proof-of-concept implementation of this algorithm in Smalltalk.

1Named after the figurehead of genetics Gregor Johann Mendel.

Given as input a source code entity that is currently being worked on by a developer, the algorithm provides the developer recommendations regarding which traits that entity may lack. The algorithm is based on a genetic-inspired metaphor. It assumes that source-code entities which are in some way related — for example by hierarchy — are often governed by the same regularities. If a particular trait, that is shared by most of its relatives, is missing from a particular source-code entity, we consider that trait as a suitable candidate for recommendation. In this way, our algorithm differs from most existing coding assistants: it does not aim at predicting suitable messages to be sent, or the next action that a developer needs to take. Rather, it merely focusses on traits that may be missing from the source-code entity, such as which methods should potentially be overridden by some class, which source-code template might be suitable for the currently browsed method, or which calls to methods or referenced types are likely missing from a method.

To validate our approach, we conducted an experiment on five open-source systems. For each of those systems, we performed a quantitative analysis of the recommendations provided by our algorithm, by comparing these recommendations with the actual implementation of the system.

This paper presents the following contributions:

- Definition of an algorithm to recommend missing traits in source-code entities, based on an analysis of related entities;
- Proof-of-concept implementation of this algorithm in the Smalltalk programming language;
- Proposal of an automated approach to validate such recommendation systems;
- Quantitative validation of the recommendation algorithm on five Smalltalk systems using the proposed approach;

The remainder of this paper is organized as follows: Section 2 presents a usage example of Mendel. Section 3 explains in detail the metaphor and the approach used to implement it. Section 4 describes the method used to validate the results of the approach. Section 5 presents the results and discusses them, as well as threats to validity. Section 6 compares our approach with related work and Section 7 summarizes the conclusions of our work.
II. MENDEL’S USAGE

We start our description of Mendel by detailing a typical usage scenario. The example is taken from the implementation of IntensiVE, which also serves as one of the five case studies used to validate our approach (see Section V). IntensiVE is an academic tool suite for co-evolving design documentation and source code. It implements various user actions and undo functionality using the Command design pattern [3]. Each user action is modeled as a separate class inheriting from IVIntensiVEAction. These subclasses must override the methods performAction and name in order to, respectively, specify the behavior associated with the action, and return the name of the action as it will appear in the interface.

Suppose now that a developer wishes to add a new ‘remove’ action to the system. The developer does so by creating a new subclass RemoveAction of IVIntensiVEAction. Mendel provides two categories of recommendations: traits that the new class must exhibit and traits that it may exhibit. For example, it informs the developer that the new class must implement methods performAction, undoAction and name, and that it should contain the identifier ‘Action’ in its name.

After having added the new class, the developer starts implementing the method name. Mendel will now recommend that this method should be classified in the protocol ‘accessing’ and follow a source-code pattern that returns a string. The developer updates the method such that it does so.

Next, the developer adds a method performAction. Mendel provides him with a list of methods that are potentially useful to be invoked. For example, he may wish to call the triggerEvent method that notifies the user interface to update after the model changed. Furthermore, he is reminded that similar methods make use of the class Factory, and that these methods perform a super call.

Although not all source code required to complete the new action class is predictable, our tool can guide the completion by suggesting common implementation traits in actions such as names, calls, references, and method structures/templates.

III. MENDEL’S APPROACH

Mendel aims to detect missing traits in source code entities by analyzing how they differ from source-code entities in their vicinity. In what follows we provide an overview of the algorithm that Mendel uses to provide developers recommendations for a source code entity that is currently being worked on. The approach relies on the assumption that it is possible to provide suggestions on what is missing in the chosen entity by analyzing entities related to it [4]. These related entities can be regarded as “family” of the browsed entity, which inspired us to use a genetic metaphor. Our approach consists of 5 steps:

1) Given some source-code entity, retrieve that entity’s family, i.e. all entities that are closely related to it;

2) For the selected entity and each of the entities in its family, calculate their traits: i.e. the set of properties that characterize those entities;

3) Identify the dominant family traits, i.e. those traits shared by most members of the family;

4) Identify the recessive family traits, i.e. those traits exhibited by a significant, but non-dominant, subset of the family of traits;

5) Propose missing traits: traits the selected entity must exhibit (i.e. the dominant traits of the family that are not present in the entity) or traits that may be of interest to the developer (i.e. the recessive traits).

Below, we explain each step in detail, using a running example which is a reduced version of the example presented in Section II. In particular, consider the class hierarchy displayed in Figure 1. This hierarchy consists of a number of classes that implement a Command design pattern.

![Diagram](image)

Suppose now that a developer adds to this hierarchy a new class named RemoveAction (indicated with dashed lines), and wishes to implement the method performAction. In what follows, we describe each of the steps of our algorithm to compute the set of recommendations for implementing that method performAction.

a) Retrieve the family of the analyzed entity: As a first step of our algorithm, we start by retrieving the set of source-code entities related to the entity that is currently being worked on. Identifying this set of family members family(e) relies on the assumption that we can provide useful recommendations based on closely related source-code entities. Mendel allows for the analysis of either classes and methods.

Since we analyze object-oriented systems, we consider the family of a class to be all classes in the same hierarchy. This set of family members is computed by taking the direct superclass of the selected class and returning all of this superclass’ direct subclasses, and the subclasses of these direct subclasses, except for the class analyzed (the class analyzed is excluded from the family). In other words, the family of a class are its siblings and nieces.

Furthermore, while determining the family of a class, we make a distinction between whether the analyzed class is concrete or abstract. If the developer is working on an abstract
class, then the recommendations obtained by analyzing related abstract classes will be more relevant than those from concrete classes. Therefore, we restrict the family of an abstract class to classes that are also abstract, whereas the family of a concrete class will contain only concrete classes. When the analyzed class does not belong to any class hierarchy (i.e., it directly inherits from `Object`) its family is empty.

The family of a method is defined as the set of all methods with the same signature, within classes of the family of the method’s implementing class. For example, the result of calculating the family of the method `performAction` of class `RemoveAction` is illustrated in Figure 2. In this figure, the family members of the selected entity are shown in black. All other entities have been grayed out. In our calculation, we consider all direct subclasses of `ApplicationAction` (the direct superclass of `RemoveAction`), except the abstract class `AbstractClipboardAction`, and their subclasses, to be the family of `RemoveAction`. All implementations of the method `performAction` in any of these classes are considered to be family members of the entity.

b) Find the traits of the source-code entity analyzed and of its family members: The second step of our algorithm consists in determining the traits `traits(e)` of the source-code entity analyzed, and of that entity’s family members. For a class we consider the following properties:

- The keywords that compose the class’ name. For the class `ApplicationAction` these are ‘Application’ and ‘Action’;
- All ancestors of the class;
- The signatures of all methods implemented by the class;
- All types referred to from within the class.

For methods we calculate the following properties:

- A generalized parse tree of the method, giving an abstract representation of the method’s structure. A generalized parse tree matches a piece of source code if, except for the literal values and a renaming of variables, it matches the parse tree of that source code;
- All types used by the method;
- The signatures of all methods invoked from within the method;
- The protocol in which the method is classified;
- All super calls occurring within the method.

To illustrate the idea, Figure 3 shows a (fictitious) list of traits for each of the `performAction` methods in the family of `RemoveAction.performAction`.

![Fig. 2. Determining the family of performAction.](image1)

**Fig. 2.** Determining the family of `performAction`.

![Fig. 3. Determining the traits of the family members of performAction.](image2)

**Fig. 3.** Determining the traits of the family members of `performAction`.

c) Find the dominant traits in the family: The third step of our algorithm consists in identifying the dominant traits `dominantTraits(e)` that characterize the members of the family. Dominant traits are those that are exhibited by most of the entities of the family. At first glance, it might appear logical to consider traits to be dominant only if they are shared by all family members. From previous experience [6] however we observed that regularities tend to be not uniformly respected in the source code: while a majority of the methods in a family may respect for example a naming convention, it is not necessarily the case for all. Such deviations are typically caused by the fact that regularities are often only implicitly known and are not automatically enforced in the source code.

It is to accommodate such deviations that we consider a trait dominant if the majority of the entity exhibits this trait.

![Fig. 4. Dominant traits in the running example.](image3)

**Fig. 4.** Dominant traits in the running example.

As sizes of families tend to vary, the majority threshold was chosen in such a way that it allows for deviations even for small families, but such that the number of deviations is still proportional to the size of the family. After a trial-and-error validation we found that a log₂ of the size of the family behaved well for most of the families tested. In other words,
for a given family of entities \( F \), we consider a trait dominant if at least \( \tau_d(F) \) members of the family exhibit that trait, where the threshold function \( \tau_d \) is defined as:
\[
\tau_d(F) := \left| F \right| - \left\lfloor \log_4 \left| F \right| \right\rfloor.
\]

Figure 4 shows the dominant traits of the performAction methods in our running example. As all performAction methods perform a super call and call a method triggerEvent, these traits are dominant for method performAction. Furthermore, all methods, except the method performAction on class PasteAction, are classified in the protocol ‘operations’. Within our running example, the value of \( \tau_1 \) is 4 (there are 5 elements in the family \( F \) of method performAction: \( \left\lfloor \log_4 \left| F \right| \right\rfloor \) equals 1). Since the amount of entities in the family exhibiting this property is still larger than or equals than 4, the property of being classified in the protocol ‘operations’ is also considered a dominant trait.

d) Find the recessive traits: Recessive traits are used to detect traits that might be needed by the analyzed entity but that are shared only with a smaller subset of the family. That is, recessive relatives share characteristics with the entity analyzed that are beyond obvious family characteristics (i.e., dominant traits).

We define the set of recessive traits of an entity \( e \) as those family traits that are not part of the dominant traits of the entity, but that are present in at least \( \tau_r \) members of the family. Again by trial-and-error (analyzing the Pier system that also serves as one of our case studies, see Section IV), we came up with the following definition for \( \tau_r \):
\[
\tau_r(e) = \begin{cases} 
\frac{3}{2} \cdot \left| \text{family}(e) \right| & \text{if } e \text{ is a class} \\
\frac{1}{2} & \text{if } e \text{ is a method}
\end{cases}
\]

For a class, we consider a trait to be recessive if it is shared by at least two thirds of the family members. As the number of family members of a method often tends to be much smaller than that of a class, we consider a method trait recessive as soon as it is shared by 2 of the method’s family members.

![Fig. 5. Recessive traits in the running example.](image)

If we apply this definition to our running example (also see Figure 5), we identify the trait that performAction methods use the type ActionStack and call the methods updateParent and add as recessive traits, as they are shared by three of the family members of the method we are analyzing.

e) Propose the suggested traits for the analyzed source-code entity: The proposed traits are suggested changes or additions to the source code entity analyzed. As explained before, we distinguish between two kinds of proposed traits: traits that the analyzed entity probably\(^3\) must exhibit, and traits that the entity may want to exhibit. This distinction is made to emphasize the fact that a different level of certainty is associated with these recommendations.

We consider traits that are not present in the analyzed entity, but that are dominant with respect to the family, as properties that very probably should be implemented by the entity. We define this set of traits as:
\[
suggestedMust(e) = \text{dominantTraits}(e) \setminus \text{traits}(e)
\]

Applied to our running example, Mendel would suggest that the method performAction must perform a super call, call the method triggerEvent and be classified in the protocol ‘operations’.

Conversely, traits shared by recessive relatives of an entity \( e \) are considered as traits that the entity might exhibit. This set of traits is defined as:
\[
suggestedMay(e) = \text{recessiveTraits}(e) \setminus \text{traits}(e)
\]

In our running example, our tool proposes that the method may refer to ActionStack and call updateParent or add.

The set of all suggestions is defined as:
\[
suggested(e) = \text{suggestedMust}(e) \cup \text{suggestedMay}(e)
\]

IV. Empirical validation

As a validation of our approach, we present a quantitative analysis of the recommendations proposed by our Mendel tool. The validation has as goal to assess whether Mendel is able to correctly suggest missing traits for a given source-code entity.

A. Experimental setup

To measure the correctness of Mendel’s recommendations, we propose to compare the traits it suggests with those that are actually present in the source code. The idea behind our experimental setup is to take the source code of a program, remove the implementation of a particular source-code entity, and then check whether or not Mendel’s suggestions align with the original properties of that entity. Source code removal a standard approach to validate the quality of recommendations.

\(^3\) Note that even for the dominant traits there is no guarantee, only a high likelihood, that the analyzed entity should indeed exhibit that property, but in the end it is up to the developer to make the final decision.
While this experiment does not provide any insights into whether the recommendations of the tool are actually useful to developers, it does allow us to assess to which degree the traits suggested by the tool are correct with respect to the original implementation. We implemented a small tool to fully automate this validation process:

1) As input, this validation tool is offered the source code of a particular software system;
2) For each source-code entity (classes and methods) in that system, it computes (using Mendel) and stores the traits that are exhibited by those source-code entities;
3) It then iterates over all source-code entities in the system again and:
   a) removes the implementation of that source-code entity. In the case of a class, all of its methods are removed; in the case of a method, its method body is removed;
   b) uses Mendel to suggest missing traits of the entity;
   c) compares these recommended missing traits with the original traits of the entity;
   d) restores the original implementation of the entity.
4) Finally, it generates a report summarizing the results of the validation.

To validate whether our approach correctly suggests missing traits, we defined a number of metrics that use the intermediate data collected via the validation process detailed above. We sketch these measurements using the Venn diagram shown in Figure 6.

![Venn diagram](image)

The Venn diagram shows four sets: \( \text{traits}(e) \), \( \text{predictable}(e) \), \( \text{suggested}(e) \) and \( \text{correct}(e) \). The set \( \text{traits}(e) \) contains all traits of the original (before its implementation was removed) source-code entity \( e \) that is being analyzed, and that were stored in step 2 of the process described above. \( \text{suggested}(e) \) contains all traits of the entity \( e \) that were recommended by Mendel, as explained in the previous section. \( \text{correct}(e) = \text{traits}(e) \cap \text{suggested}(e) \) is the subset of suggested traits that are indeed traits of the original entity. \( \text{predictable}(e) \) is a subset of \( \text{traits}(e) \) which contains all traits of the entity that are shared by at least one other source-code entity. The set \( \text{predictable}(e) \) thus contains all traits of entity \( e \) that we can expect or hope our tool to recommend. Idiosyncratic traits that only appear in \( e \) but nowhere else are impossible to suggest after \( e \)’s implementation was removed, because removing \( e \) also removes its traits from the system. We will therefore not consider such idiosyncratic traits as false negatives, but restrict our metrics to the predictable ones.

Using the four sets defined above, we define the following metrics for the recommendations that our tool suggests for a particular entity \( e \):

- **Precision** is defined as the ratio of correct recommendations over the total number of recommendations and is a measure of how correct the suggestions provided by the tool are:

\[
\text{precision}(e) = \frac{|\text{correct}(e)|}{|\text{suggested}(e)|}
\]

- **(Original) recall** is the ratio of correct recommendations over the total number of original traits and is a measure of how many correct traits the tool has missed:

\[
\text{originalRecall}(e) = \frac{|\text{correct}(e)|}{|\text{traits}(e)|}
\]

- **Predictable recall** expresses the recall of our tool with respect to the set of predictable properties. We define it as the ratio of correct recommendations over the number of predictable traits:

\[
\text{predictableRecall}(e) = \frac{|\text{correct}(e)|}{|\text{predictable}(e)|}
\]

### B. Analyzed systems

We validated our tool on five different open-source Smalltalk systems. Table I gives an overview of these systems. The systems vary in terms of size (from less than 200 classes to over 600 classes), and stretch over various domains (content management, academic tools, web framework, graphics).

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Classes analyzed</th>
<th>Methods analyzed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pier</td>
<td>Content management system</td>
<td>184</td>
<td>389</td>
</tr>
<tr>
<td>IntensiVE</td>
<td>Academic software engineering tool</td>
<td>122</td>
<td>317</td>
</tr>
<tr>
<td>MOOSE</td>
<td>Academic reverse engineering env.</td>
<td>159</td>
<td>348</td>
</tr>
<tr>
<td>Seaside</td>
<td>Framework for web applications</td>
<td>344</td>
<td>854</td>
</tr>
<tr>
<td>Jun</td>
<td>3-D graphics library</td>
<td>603</td>
<td>4402</td>
</tr>
</tbody>
</table>

### V. Results

Table II provides an overview of the average amount of suggestions per entity our tool proposes, and the time necessary to complete the entire validation experiment per system\(^4\). On average, the number of recommended traits per entity remains very low: around 5 for IntensiVE; around 3 for all other systems. If we compare the number of recommendations at the level of classes and at the level of methods, we see that

\(^4\)The closer the recall is to zero, the more false negatives there are, whereas a recall of 1 indicates that all traits of the entity were correctly found.

\(^5\)When executed on an Apple Mac Mini with a dual-core 2.4Ghz Core2 Duo processor and 2Gb of RAM
there are no significant differences. Nevertheless it seems that the tool finds slightly more recommendations for methods than for classes. The time necessary to automatically execute these validation experiments ranged from between less than 1 minute (for the whole of Pier) to 6 hours (for the whole of Jun). While not shown, depending on the class and system chosen, Mendel took between 1 and 43 seconds to analyze a single class with its corresponding methods.

Figure 7 provides an overview of the precision of Mendel when applied to each of our five case studies. To visualize our results, we use violin plots. While the top row of this figure details the precision of our tool for class recommendations, the bottom row contains the results for methods.

The left-most violin plot on the top row details the overall precision of the suggestions for classes for each of the systems we analyzed. As we can see, for all systems – with the exception of Moose – our approach scores reasonably well with the median precision being 100% and at least 75% of the suggestions for classes having a precision above 50%. Except for Moose’s distribution of precision whose class recommendations are correct around 50% of the time, most of the recommendations for classes tend to have a precision higher than 70% (nail-shaped violins, with crests on the top).

Remember from above that our approach classifies the proposed missing traits in two categories, namely those that the browsed entity must exhibit and those that it may exhibit. The middle and right-most violin plots on the first row of Figure 7 drill down our analysis to these two categories. The may precision tends to outperform the must precision for classes in all applications except Moose. In particular, all classes in IntensiVE and Seaside with may recommendations had all recommendations correct, while all classes in Moose with may recommendations had only 40% correct recommendations.

The bottom row of Figure 7 shows the precision of the recommendations for the methods of the software systems we analyzed. The median precision for method recommendations is above 80% for Seaside and Pier, and just below 70% for Moose and IntensiVE. Nevertheless, this value is always above 60% regardless of the system analyzed. Note that some of the violins – except Pier and Jun’s – have the shape of an elongated eight. This means that there is a group of methods for which recommendations tend to be largely correct (more than 80%; the crest on the top of the graph) and a group of methods for whose recommendations tend to be incorrect (the crest on the bottom of the graph).

If we distinguish between must and may, we can notice that our tool performs much better for providing must recommendations for methods, having a median precision of 66% for Moose and 100% for the other case studies. For all systems except Moose, the recommendations for methods score for a large part of the analyzed methods above 80%. We can see that the precision of the traits our tool proposes as a must outperforms the traits proposed as a may — contrary to our observation with classes — when we compare the precision of the must and may. For the method traits our tool recommends as a may, the median precision lies around 30%, which is considerably lower than the one of classes. Notice that the median precision for Pier is 0%, while for Moose it is approximately 50%.

Figure 8 summarizes the recall of our approach with respect to respectively all original properties of the entity (left violin plots of Fig. 8) and to the predictable properties of the entity (right violin plots of Fig. 8). As we can see, the median of the recall lies between 50% and 70%. Although the median of the recall remains within the same range for both figures, when the recall is calculated with respect to the original properties it tends to be lower than when it is calculated with respect to the predictable properties. Nevertheless Pier (classes and methods), Moose (methods) and Jun (classes) seem to be less susceptible to idiosyncratic traits as their median does not change much between the left and right violin plot. With the exception of IntensiVE, which appears to score better than the other case studies, the distributions of the recall look similar for all case studies.

A. Discussion

In what follows, we present a discussion of the results we obtained, and identify some of the advantages and limitations of Mendel. Based on these observations, we identify some future work.

1) Interpretation of the results: The experiment reported seems to indicate that our approach works reasonably well as the precision lies acceptably high both for classes and for methods. The results show that Mendel is able to correctly recommend the traits that were present in the original class or method. Both for classes and methods, the recommended traits

### Table II: Results of our experiment

<table>
<thead>
<tr>
<th>System</th>
<th>Avg. suggested</th>
<th>Avg. suggested (classes)</th>
<th>Avg. suggested (methods)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pier</td>
<td>3.2</td>
<td>2.4</td>
<td>3.6</td>
<td>44s</td>
</tr>
<tr>
<td>IntensiVE</td>
<td>5.2</td>
<td>4</td>
<td>5.6</td>
<td>244s</td>
</tr>
<tr>
<td>Moose</td>
<td>3.7</td>
<td>4.2</td>
<td>3.5</td>
<td>260s</td>
</tr>
<tr>
<td>Seaside</td>
<td>3.0</td>
<td>1.4</td>
<td>3.7</td>
<td>580s</td>
</tr>
<tr>
<td>Jun</td>
<td>3.7</td>
<td>3.4</td>
<td>3.7</td>
<td>6h</td>
</tr>
</tbody>
</table>
Our results are consistent over the different case studies with the exception of Moose, which scores significantly worse than the other case studies from the point of view of precision. One possible explanation for this can be found if we analyze the size of families for the entities that were analyzed. A summary of these family sizes can be found in Figures 9 (for classes) and 10 (for methods). As we can see, the family size of entities in Moose tends to be lower than for the other case studies (for Pier, the families are also quite small, but Pier is the smallest system analyzed).

Especially for methods, the family sizes in Moose are low; this might be a good indicator why our approach exhibits lower precision for recommended method traits in Moose. Overall, it is not surprising that the precision of our approach is coupled to the size of the hierarchies: the larger the class hierarchy is of an entity that is being analyzed, the larger that entity’s family will be and the more accurate our recommendations will be. Conversely, when applied to a system with small class hierarchies, our approach will not be able to provide recommendations.
Our experiment also illustrated that method traits recommended as may perform worse than method traits recommended as must. While this might appear to be a problem that hampers the precision of our approach, in practice, we found that our approach presents the user with far fewer mays than musts. Table III gives an overview of the ratio of musts with respect to the total number of recommendations. For Pier and Moose, practically all recommendations that our tool proposes are must. For Seaside and Jun, this lies around 70% of the recommendations. Only for IntensiVE, this number lies lower, and there exists a large difference between classes and methods. One possible explanation for this is that the families of methods of IntensiVE tend to be larger than with the other case studies (the median of Fig. 10 lie higher than with the other case studies). Consequently, the chance that a method is shared by two or more family members is larger than with the other case studies. While they are a minority, we decided not to prune traits classified as mays as they still can convey useful information.

2) **Usefulness of Mendel’s results:** The experiment described above only provides insights into the extent to which our approach is able to provide correct recommendations. As such, we cannot make any assumptions regarding the usefulness and usability of our approach, which would require us to conduct a user study (which lies outside the scope of this paper). In particular, our tool differs from other related tools in that it does not aim at providing developers a suggestion what to do next, but rather offers developers a list of traits that – according to our tool – are missing from the currently browsed entity. While we plan to further enhance our tool such that it does not only provide recommendations, but also will aid developers in exploring the rationale of each recommendation, we still expect a user of our tool to manually interpret the results that our tool proposes. Although we cannot make any claims regarding the usability of our approach, we are fairly optimistic. For all five case studies reported on above, the average number of recommendations remained quite low (less than 5 recommendations on average). As such, we feel that the overhead associated with manually inspecting these recommendations will be limited to a minimum.

3) **Selection of traits:** Our current implementation of Mendel focusses on analyzing purely structural traits: which messages get sent, which types are referenced, which structure does a method have, and so on. This set of traits is by no means complete: we can think of other traits to include such as annotations, the order in which statements occur, pair-wise presence of method calls (e.g., open and close), etc. While it is tempting to just add new traits to Mendel, we have to be careful about this as it might have a potentially negative impact on the precision and recall of our approach, and in particular on the number of recommendations that our tool proposes.

4) **Selection of family:** While not reported in the paper, we have experimented with different kinds of definitions for the family of a class. Amongst these experiments were only considering sibling classes, considering the entire class hierarchy, and so on. While only selecting the sibling classes allowed us to slightly increase the precision of our approach, it had a detrimental effect on recall. Our current definition of taking all siblings and nieces of the class provides a balance between precision and recall.

As we already mentioned earlier, our approach relies on the assumption that, by analyzing the related entities in the hierarchy in which a source-code entity is present, we can provide recommendations regarding missing traits of that entity. Consequently, if the entity we analyze is not in a class hierarchy (it is direct subclass of object), or when the size of the hierarchy is small, our tool is not able to provide (correct) recommendations. One way of circumventing this limitation of our approach is to investigate other ways of defining the family of a source code entity: for example, we could define the family of a class by considering all classes in the system that have a similar name, by analyzing comments, versioning information, and so on. Care should be taken however as such alternative definitions may impact the precision and recall of our approach.

**B. Threats to validity**

A number of factors may impede the validity of the experiment described above. First, all five selected case studies are open-source systems implemented in Smalltalk. This selection may therefore not be general enough to make any claims about the applicability of our approach to other cases. While we have selected as case studies systems ranging over different domains (academic software engineering tools, web application, frameworks), further study using systems implemented in other programming languages, and using industrial source code would be required to make well-founded claims regarding the applicability of our tool on such cases.
Second, the thresholds used by our tool were determined experimentally by means of the Pier system. While using the same thresholds yields similar results for all case studies, there are no guarantees that these thresholds can be generalized with respect to other systems regardless whether they are implemented in Smalltalk or any other language, or when analyzing different kinds of traits for classes and methods.

Third, our validation evaluates whether recommendations are correct by comparing them with the traits that were originally present in the source code. However, if the original source code was either incorrect, or did already follow its own regularities strictly, this will have an impact on precision and recall of our approach.

Finally, while the traits we analyzed – with the exception of method protocols – are not specific to Smalltalk, precision and recall for these traits might not yield similar results when applied to other systems.

**DOWNLOAD AND ACCESS TO EXPERIMENTAL DATA:**

Our research prototype Mendel is available on [http://soft.vub.ac.be/mendel](http://soft.vub.ac.be/mendel). All experimental data that is reported on in this paper can also be found on that web page.

**VI. RELATED WORK**

In previous work [6], we mined for architectural knowledge embedded as structural regularities shared by the source code entities of an application. Such work was used as input for recommendation tools like MEntoR\(^ {10}\) [9] that aimed at showing conformance of source code structural regularities of the application, and Clairvoyant\(^ {11}\) that aimed at showing characteristics that a source code entity might require. Contrary to these tools, Mendel extracts context-dependent regularities, which allows the developer to exploit regularities in a timely manner.

Table IV summarizes approaches that mine source code to give recommendations and are therefore related to our tool Mendel. All of these tools share the common goal of facilitating the usage of third party code (like APIs and frameworks). In the table, we show for each approach the kind of data that is used as input, the granularity of the provided recommendations, a brief description of the technique that is applied to identify recommendations, and whether or not a corpus is necessary to train the technique.

From all techniques shown in the above table, Design Prompter [11] and FrUiT [10] are closest related to our approach. Similar to Mendel, they propose missing information for a particular source-code entity. Design Prompter builds a corpus of classes implementing methods with similar signatures. It finds similar classes – based on the name of the class currently browsed and the signatures of the methods it implements – and proposes missing methods [11]. FrUiT builds a corpus of association rules of structural properties in files that use a framework, filters those rules that share structural properties with the browsed entity, and proposes structural properties that are missing from the browsed entity [10]. Our approach differs in the knowledge base used; while Design Prompter analyzes a vast number of open source repositories and FrUiT requires a set of examples using the framework analyzed, Mendel only looks at regularities in the local code base. The fact that Mendel’s knowledge come from the local code base makes it complementary for ensuring local implementation restrictions and code idioms of a company or of a team. Besides, Mendel’s recommendations are more diverse than those of Design Prompter because it suggests changes beyond missing methods, and those of FrUiT because Mendel’s recommendations go beyond framework usage. Finally, Mendel calculates the recommendations each time it is called by extracting the structural properties in the latest version of the code base. This has two advantages: Mendel does not require to analyze a huge dataset before being able to make recommendations, and it can accommodate the recommendations to the latest changes in the source code.

Tools such as Strathcona [12] recommend developers examples of source-code entities that are similar to the entity that is currently browsed. To use this information, developers need to map these examples to the current task at hand. First, this process can be error-prone if the examples are simply copy-pasted [14]. For example, if a developer omits to rename a variable, this can result in erroneous behavior. Second, to identify which of the traits of an example are relevant to the current context, a developer needs to manually analyze the recommended examples. If the developer is not aware which constraints are governing the source code, identifying which traits are applicable can be difficult [15]. To alleviate this problem, tools such as Jigsaw [16] have been proposed, that aid developers in identifying which parts of a proposed example are relevant. Our approach on the other hand proposes users a set of traits that are missing from the currently browsed entity, thereby reducing the effort required to assess which properties are relevant. Furthermore, this process is aided by the fact that Mendel classifies missing traits as either must or may, thereby providing users clues regarding how relevant the proposed traits are.

Furthermore, there exist a number of tools that, taking into account the current method context, propose which method to call next. Bruch et al. [8] proposed the concept of Best Matching Neighbors, which is similar to our concept of family. The Best Matching Neighbors is the set of classes with a similar set of method calls as the one in the scope in which the code completion is called. This approach aims at filtering irrelevant proposals of code completion suggestions using the confidence of other methods called in the set of Best Matching Neighbors. RASCAL [7] provides similar functionality by analyzing the similarity between classes based on the methods that are called. This tool uses the previous called method, to predict which method needs to be called next. The MAPO tool by Xie et al. [13] proposes developers which sequence of methods should be called, based on the currently browsed source-code entity. While these approaches provide recommendations at a more fine-grained level than our approach, they are limited to

---

\(^{10}\)[http://www.info.ucl.ac.be/~alozano/mentor.html]

\(^{11}\)[http://www.info.ucl.ac.be/~alozano/clairvoyant.html]
proposing only a single kind of trait, in contrast to Mendel which proposes various kinds of missing traits.

VII. CONCLUSION

In this paper we introduced Mendel, a tool for recommending missing traits for classes and methods. Mendel is based on the assumption that related source-code entities often share similar traits (naming conventions, structural patterns, and so on). Our tool leverages this information to propose traits that may be missing in the currently browsed entity, but that are present in related source-code entities.

To provide a quantitative validation of our approach we conducted an experiment in which we applied our tool to five open-source Smalltalk systems. For each system, we compared the recommendations proposed by our tool with the traits originally present in the source code. This experiment allowed us to observe that Mendel can provide recommendations with good precision and acceptable recall if the family of the analyzed source-code entity is sufficiently large.

As for future work, we plan to extend our approach to support other programming languages, analyze other kinds of traits and experiment with various ways of defining the family of a source-code entity.

ACKNOWLEDGMENTS: Angela Lozano is funded as a post-doc researcher on a an FNRS-FRFC project. This work has also been supported by the Interuniversity Attraction Poles (IAP) Programme of the Belgian State – Belgian Science Policy.

REFERENCES

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

TABLE IV
COMPARISON OF SOME APPROACHES THAT ANALYZE SOURCE CODE TO RECOMMEND HOW TO COMPLETE A PROGRAMMER’S CODE

<table>
<thead>
<tr>
<th>Tool</th>
<th>Data used</th>
<th>Recommendation granularity</th>
<th>Recommendation principle</th>
<th>Corpus required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mendel</td>
<td>Structural description, class &amp; method names</td>
<td>Implementation hints</td>
<td>Similarity between methods or classes based on their structural facts (i.e., traits)</td>
<td>No</td>
</tr>
<tr>
<td>FrUIT [10]</td>
<td>Structural description</td>
<td>Implementation hints</td>
<td>Association rules that contain structural properties in the browsed file</td>
<td>Yes</td>
</tr>
<tr>
<td>Design Prompter [11]</td>
<td>Signatures of methods</td>
<td>Set of methods missing</td>
<td>Number of similar signatures shared by the user’s class and the classes in the corpus</td>
<td>Yes</td>
</tr>
<tr>
<td>Strathcona [12]</td>
<td>Structural description</td>
<td>Examples of how to complete a method</td>
<td>Similarity between methods based on different heuristics per type of structural fact</td>
<td>Yes</td>
</tr>
<tr>
<td>Best Matching Neighbors [8]</td>
<td>Called methods</td>
<td>Next method to call</td>
<td>Percentage of times both methods have been called together</td>
<td>Yes</td>
</tr>
<tr>
<td>RASCAL [7]</td>
<td>Called methods</td>
<td>Next method to call</td>
<td>Similarity between classes based on methods called</td>
<td>Yes</td>
</tr>
<tr>
<td>MAPO [13]</td>
<td>Called methods, class &amp; method names</td>
<td>Sequence of method calls</td>
<td>Similarity between classes based on methods called</td>
<td>Yes</td>
</tr>
</tbody>
</table>