Mendel: Source Code Recommendation based on a Genetic Metaphor

Angela Lozano
Université catholique de Louvain
angela.lozano@uclouvain.be

Andy Kellens
Vrije Universiteit Brussel
akellens@vub.ac.be

Kim Mens
Université catholique de Louvain
kim.mens@uclouvain.be

***Abstract*—When evolving 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?" or "Should my implementation follow particular naming or structural conventions?". 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. An initial validation of Mendel seems 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. 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 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 Mendel[1], along with a proof-of-concept implementation of this algorithm in Smalltalk. 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 class 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 a 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.

[1]Named after the figurehead of genetics Gregor Johann Mendel.

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, 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 that a developer wishes to add a new 'remove' action to the system. The developer does so by creating a subclass `RemoveAction` of `IVIntensiVEAction`. Mendel provides two categories of recommendations: traits that the new class *probably* must exhibit and traits that it *possibly* may exhibit. For example, it informs the developer that the new class probably should 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 probably be classified in the protocol 'accessing'[2] 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

[2]Smalltalk allows methods to be annotated with the category to which they belong, called a protocol.

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. Related Work

In previous work [10], we mined the source code of software systems for architectural knowledge embedded as structural regularities shared by its source code entities. Contrary to this previous work, Mendel extracts context-dependent regularities, which allows the developer to exploit regularities in a timely manner.

Table I summarizes some 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.

IV. 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 [11]. 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. 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 classes in Figure 1, which implement a *Command* design pattern.

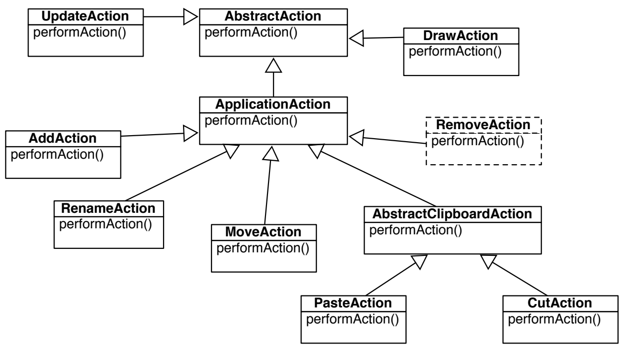

Fig. 1. Our running example.

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 e that is currently being worked on. Identifying this set $family(e)$ of family members 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 (which is excluded from the family). In genealogical terms, the family of a class are its siblings and nephews/nieces.

Furthermore, while determining the family of a class, we make a distinction between whether the analyzed class is concrete or abstract. If a 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.

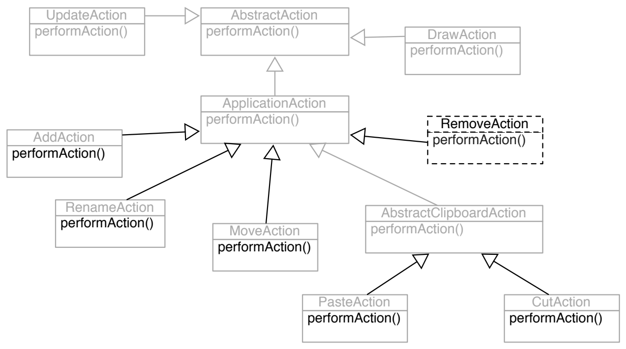

Fig. 2. Determining the family of `performAction`.

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

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

Tool	Data used	Recommendation granularity	Recommendation principle	Corpus
Mendel	Structural properties & identifiers	Implementation hints	Frequent structural properties (traits) among related methods or classes	No
FrUiT [4]	Structural properties	Implementation hints	Association rules that contain structural properties in the browsed file	Yes
Design Prompter [5]	Signatures of methods	Set of methods missing	Number of similar signatures in the user's class and the corpus' classes	Yes
Strathcona [6]	Structural properties	Examples to complete a method	Similarity between methods (different heuristics per type of structural fact)	Yes
Best Matching Neighbors [7]	Called methods	Next method to call	Percentage of times both methods have been called together	Yes
RASCAL [8]	Called methods	Next method to call	Similarity between classes based on methods called	Yes
MAPO [9]	Called methods & identifiers	Sequence of method calls	Similarity between classes based on methods called	Yes

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 e 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 list of traits for each of the `performAction` methods in the family of `RemoveAction.performAction`.

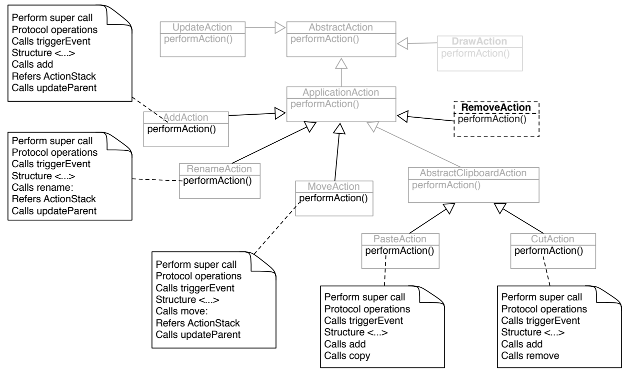

Fig. 3. Determining the traits of the family members of `performAction`.

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 of e. 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 [10] however we observed that regularities tend to be not uniformly respected in 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 family exhibits this trait.

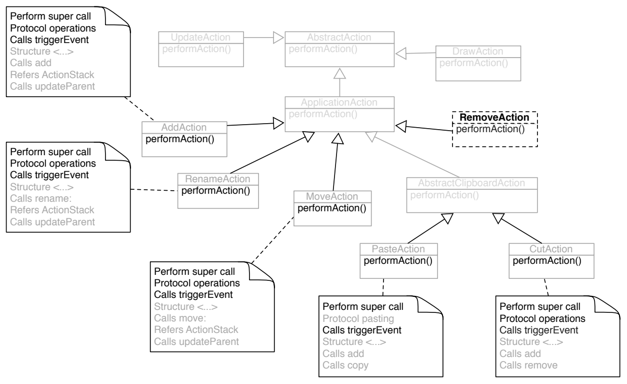

Fig. 4. Dominant traits in the running example.

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 $\log_4$ 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 $\tau_d(F) := \ |F| - \lfloor \log_4 |F| \rfloor$ family members exhibit that trait.

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 τ_d is 4 (there are 5 elements in the family F of method `performAction`; $\lfloor log_4 |F| \rfloor$ equals 1). Since the amount of entities in the family exhibiting this property is still larger than or equal to 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 *possibly* 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 $recessiveTraits(e)$ 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 τ_r members of the family. Again by trial-and-error we came up with the following definition for τ_r:

$$\tau_r(e) = \begin{cases} \frac{2}{3} * |family(e)| & \text{if e is a class} \\ 2 & \text{if e is a method} \end{cases}$$

For a class, we consider a trait to be recessive if it is shared by at least two third 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.

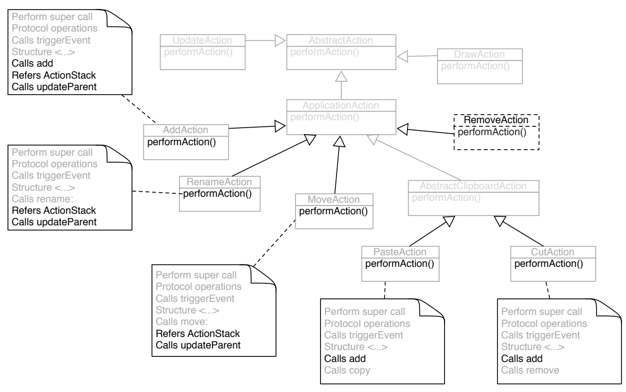

Fig. 5. Recessive traits in the running example.

If we apply this definition to our running example (see Figure 5), we identify as recessive traits that `performAction` methods use the type `ActionStack` and call the methods `updateParent` and `add`, since these traits 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 two kinds of proposed traits: traits that the analyzed entity *probably*[3] should exhibit, and traits that the entity *possibly* 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 e, but that are dominant with respect to the family, as properties that *probably* should be implemented by the entity, and define this set of traits as:

$probableTraits(e) = dominantTraits(e) \setminus traits(e)$

Applied to our running example, Mendel would suggest that the method `performAction` probably should 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 may possibly exhibit. This set of traits is defined as:

$possibleTraits(e) = recessiveTraits(e) \setminus traits(e)$

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

The set of all suggestions is defined as:

$suggested(e) = probableTraits(e) \cup possibleTraits(e)$

[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.

Acknowledgments

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

Download and Access to Experimental Data

Our research prototype Mendel is available on *soft.vub.ac.be/mendel*. To validate our approach, we performed a quantitative analysis of the recommendations provided by our algorithm, by comparing these recommendations with the actual implementation of the system. The experimental data collected on five open-source systems can also be found on that web page.

References

[1] K. Beck, *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.
[2] J. Coplien, *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.
[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
[4] M. Bruch, T. Schäfer, and M. Mezini, "FrUiT: IDE support for framework understanding," in *Proceedings of Eclipse '06 at OOPSLA*. New York, NY, USA: ACM, 2006, pp. 55–59.
[5] O. Hummel, W. Janjic, and C. Atkinson, "Proposing software design recommendations based on component interface intersecting," in *Proceedings of RSSE 2010*. New York, NY, USA: ACM, 2010, pp. 64–68.
[6] R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," *IEEE Trans. Softw. Eng.*, vol. 32, pp. 952–970, December 2006.
[7] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of ESEC/FSE 2009*. New York, NY, USA: ACM, 2009, pp. 213–222.
[8] F. Mccarey, M. O. Cinnéide, and N. Kushmerick, "Rascal: A recommender agent for agile reuse," *Artif. Intell. Rev.*, vol. 24, pp. 253–276, November 2005.
[9] T. Xie and J. Pei, "MAPO: Mining API usages from open source repositories," in *Proceedings of MSR 2006*. New York, NY, USA: ACM, 2006, pp. 54–57.
[10] A. Lozano, A. Kellens, K. Mens, and G. Arevalo, "Mining source code for structural regularities," in *Proceedings of WCRE 2010*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 22–31.
[11] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of SIGSOFT/FSE 2006*. New York, NY, USA: ACM, 2006, pp. 23–34.