An Anomaly-based Approach for Detecting Modularity Violations on Method Placement

Kazuki Yoda[1], Tomoki Nakamaru[1], Soramichi Akiyama[2], and Shigeru Chiba[1,*]
[1]The University of Tokyo, Bunkyo, Tokyo, Japan
[2]Ritsumeikan University, Kusatsu, Shiga, Japan
yoda@csg.ci.i.u-tokyo.ac.jp, nakamaru@graco.c.u-tokyo.ac.jp, s-akym@fc.ritsumei.ac.jp, chiba@acm.org
*corresponding author

***Abstract*—This paper presents a technique for detecting an anomaly in method placements in Java packages. This anomaly detection helps code reviewers discover a method belonging to an inappropriate package in modularity when developers commit changes in their software development projects. Moving such a method to an appropriate package will contribute to the maintenance of good modularity in their projects. This is particularly beneficial in the later stage of development, where modularity is often violated by adding new features not anticipated in the initial plan. Our technique is based on few-shot classification in machine learning. This paper empirically reveals that our neural network model can detect an anomaly in method placements and a significant portion of the anomalies is considered as inappropriate method placements in modularity. Our model can discover even a method placement that violates a project-specific coding rule that its developers would choose for some reason of maintainability or readability. Our technique is useful for maintaining the consistency in such a project-specific rule.**

***Keywords*—code review; refactoring; packages; Java; few-shot classification; nerual networks**

I. Introduction

The importance of software modularity has been widely recognized both in academia and industry [1]. Several modularity principles have been proposed, such as low-coupling and high-cohesion [2] and information hiding [3], and developers are recommended to follow these principles. It has been revealed that the qualities of a program will be seriously damaged if those principles are violated and program components such as functions and methods are placed in inappropriate modules [4]. The violation of those principles might cause a more serious problem such as architectural erosion [5].

A popular practice for maintaining modularity principles is code reviewing. When a developer adds or modifies a feature and commits code changes, the developer asks code reviewing by fellow, or often senior developers. The code reviewer must inspect the committed change and confirm that it does not violate modularity principles. If it violates, the reviewer recommends the developer to revise the change. However, this code reviewing is labor intensive and time consuming.

This paper presents our technique for helping code reviewers. It discovers an anomaly in method placements in Java packages. If the modularity of a project is regularly maintained by code reviewing, we can expect that a method that seems to be placed in a *strange* package is violating some modularity principles, or project-specific local coding rules for modularity, with a high probability. If the discovered anomaly actually gives a damage to modularity, the code reviewer can ask the developer to move the method into a more appropriate package so that the modularity level of the project will be preserved.

Suggesting an anomaly to code reviewers is particularly beneficial in the later stage of development lifetime. It is known that a software project is usually well modularized at the early stage of its lifetime but, as features are added and defects are repaired, the project increases in complexity and the structure of the system can deviate substantially from the original well-modularized one [6]. So we can assume that the existing code base of a project is well modularized in the early stage and an anomaly discovered in the later stage by comparing newly changed code with the rest of the existing code will be a flaw in modularity with a high probability. Fixing this flaw by moving a method will effectively contribute to keeping the modularity.

Our technique detects an anomaly in method placements in Java by few-shot classification in machine learning. We use a Java package as a module unit as in previous work [7], [8]. Our technique reports which package a method detected as an anomaly should be moved into. For few-shot classification, we adopt a neural network model known as a prototypical network [9]. We train the model with the source code of a large number of Java projects so that the model can learn how to classify given methods into appropriate packages in their project. Note that the total amount of source code of each project is too small to directly train the model with only that source code. This is why we need *few-shot* classification. The model must learn a general rule for obtaining the project-specific criteria for method classification from each given Java project. Although we are a follower of the recent trend in machine learning for software engineering tasks [10], [11], [12], [13], the use of few-shot classification/learning is new in this field as far as we know. The encoder of our prototypical network is also our own modification of the code2vec encoder [14].

This paper also empirically reveals that our technique can detect anomalies in method placements in real-world Java projects. It outperforms the existing neural-network model proposed by Liu *et al*. [15] for a similar classification task of method placements. The paper shows that a significant portion of the detected anomalies is considered as the placements of method in inappropriate packages in modularity. They

are violating some modularity principles or project-specific modularity rules. The latter rules derive from an architecture and/or a pattern adopted for the project. For example, when a project adopts the Model-View-Controller architecture, the methods for Model should belong to the model package. Note that this rule is not applicable to other projects adopting different architecture. Few-shot classification enables us to detect such anomalies. Consistently enforcing a project-specific rule to a project is desirable. Although it might not directly improve modularity, it would improve readability and thereby maintainability.

Our research contribution is to propose a new kind of technique for detecting anomalies in modularity. This would help developers consistently modularize their projects. More specifically, our contribution is two-fold:

1) We present a neural network model for few-shot classification of methods into packages in Java. It can be used to detect anomalies in method placement.
2) We empirically reveal that a significant portion of anomalies detected by the presented technique is caused by inappropriate method placements in modularity. They are violating modularity principles or project-specific coding rules for modularity. We also show that our model outperforms an existing model proposed for a similar task.

Our research artifacts including source code and trained parameters of the model are available at Zenodo [16] (https://zenodo.org/record/6367730).

II. Maintaining modularity

When a software development project is in the early stage of its lifetime and the size of its source code is still small, maintaining its modularity is relatively easy. Developers can easily preserve the module structure as originally designed. However, as the project grows, new features are added and the original structure is gradually damaged [6]. Modularity principles begin to be violated and project-specific modularity rules, if any, begin to lose their consistency.

To avoid this problem occurring in the later stage of project lifetime, before developers commit changes to the source code of a project, code reviewers inspect the changes and prevent the developers from lowering the modularity of that project in its later stage. A drawback of this widely-adopted practice is that manual inspection by code reviewers, who are often senior developers, is labor-intensive. Although the reviewers may use programming tools, such as JDeodorant [17], for measuring various source code metrics and obtaining suggestions for modularization, they must find which metrics should be used to consistently maintain the modularity rules of their project.

For example, Listing 1 is a code excerpt in Java from an Android application for photo editing available on GitHub.[1] Suppose that a developer is going to append the `rotateImage` method to this project and this change is under review. The `Editor` class is the main class of this project and its primary concern is to handle user actions on the device and maintain the view of the application. However, the `rotateImage` method, which is under review, does image processing. The place of this method seems inappropriate. Although the place might be right for some modularity principles since it is a private method invoked only within the `Editor` class, the image processing is far from the primary concern of the `Editor` class. It looks like an instance of code smell known as god object [18]. The code reviewer should advise the developer to move the `rotateImage` method into the `BitmapUtils` class in a different package `Utils`. As shown in Listing 2, that class includes several similar methods for image processing.

Programming tools can measure several source code metrics such as low-coupling and high-cohesion [2], and a distance proposed by Tsantalis *et al* [19]. Then they can suggest placing the `rotateImage` method in either the `Utils` package or the `ProfileFit` package, which contains the `Editor` class, depending on what metrics are used. Nevertheless, both packages could be a right place. The code reviewer must decide which package is a right place by considering the fact which modularity principle or rule has been applied to other methods. This fact is known only by the developers and the code reviewers of the project. To maintain good modularity in the later stage of project lifetime, a key is consistency in modularity principles or rules applied to a project. If the other methods for image processing are placed in the `Utils` package, instead of distinct packages where those methods are

[1] https://github.com/ReneGuillen/Photo_Editor_Android

```java
package com.software.ProfileFit;
public class Editor extends AppCompatActivity {
  private static Bitmap rotateImage(
      Bitmap img, int degree) {
    Matrix matrix = new Matrix();
    matrix.postRotate(degree);
    Bitmap rotatedImg = Bitmap.createBitmap(
        img, 0, 0, img.getWidth(),
        img.getHeight(), matrix, true);
    img.recycle();
    return rotatedImg;
  }
  // more code here
}
```

Listing 1: A main class with very detailed implementation

```java
package com.software.ProfileFit.Utils;
public class BitmapUtils {
  public static Bitmap bitmapOverlayToCenter(
      Bitmap bitmap1, Bitmap bitmap2,
      Bitmap bitmap3) {...}
  public static Bitmap applyOverlay(
      Context context, Bitmap sourceImage,
      int overlayDrawableResourceId) {...}
  public static Bitmap blurRenderScript(
      Editor context, Bitmap smallBitmap,
      int radius) {...}
  private static Bitmap RGB565toARGB888(Bitmap img)
      throws Exception {...}
}
```

Listing 2: Most image processing methods are included in this class

```
package com.service;
public class AuthorService_mybatis {
  public Author findAuthor(Long id) {
    return this.authorMapperMybatis.findAuthor(id);
  }
}
```

Listing 3: A concrete method in the `service` package

called, `rotateImage` should be also placed in `Utils`.

Listing 3 is another example. This is an excerpt from a project for a web service available on GitHub[2]. It is a tiny project for practicing the development using the Spring framework[3] in Java. Thus, the project is based on the Model-View-Controller (MVC) architecture. It also uses the MyBatis framework[4] for database access. Suppose that we are reviewing the `findAuthor` method declared in the `service` package, which is the module for the model of the MVC. This looks fine because `findAuthor` obviously belongs to the model of MVC, but other similar methods are in the `service.impl` package. Only interfaces are included in the `service` package. Hence, the whole declaration of the `AuthorService_mybatis` class should be moved into the `service.impl` package. It is a common practice to place concrete classes and methods into an implementation package and separate them from their abstract interfaces in another package.

If we know that an interface and its implementations are placed in separate packages in this project, it would be easy to automate the detection of `findAuthor` as a method placed in an inappropriate package. However, this project may not adopt this modularity rule and the current `service` package may be a right place to put `findAuthor`. To decide the right package for `findAuthor`, we must investigate other methods to see how a package is selected for placing a method. Again, this depends on which modularity rules have been adopted for the project.

III. Detecting anomalies in method placement

We present a technique for detecting anomalies in method placement in Java that will help code reviewers judging whether a method belongs to an appropriate Java package. Our technique would contribute modularity maintenance by code reviewers. As mentioned in the previous section, which package a given method should belong to depends on how other methods have been placed in packages in their project, particularly in the latter stage of project lifetime. If a method under review is placed in an anomalous place, the probability of modularity violation by this placement would be high.

Our technique uses a machine learning model that has been trained to classify a method into the package to which it belongs. In order to detect an anomaly, our technique first tells the model to infer to which package a method belongs. If the inferred package is not the package to which the method actually belongs, our technique reports such method placement as an anomaly.

The machine learning model we use is a prototypical network [9]. A prototypical network consists of an encoder and classifier. The encoder is a neural network that encodes source code into a distributed (vector) representation. The classifier is a component that outputs to which package a method is likely to belong. The encoder (neural network) has learnable parameters. The classifier, on the other hand, does not have learnable parameters.

Figure 1 illustrates how we use a prototypical network in our anomaly detection. A prototypical network takes two sets of methods as its input: support set and query set. A support set is a set of methods that is composed by sampling several methods from all packages. A query set is a collection of methods that the prototypical network infers packages for. In the inference step, the encoder converts each method in the support/query set into a distributed representation. The classifier computes the probability of being classified into each package. The probability is computed based on the distance between distributed representations. When the representation of the query method is closer to the cluster for a package, the classifier outputs a higher probability for that package.

The prototypical network is trained with a large corpus of Java projects. Specifically, we (1) build support/query sets for each project in the corpus, (2) feed them to the prototypical network, and (3) update learnable parameters of the encoder so that the network shows a good classification performance. Note that, in each iteration, the network performs the method classification task in a situation where only a small number of sample methods are given to the network. By updating the parameters of the encoder, the network is optimized for such a classification task.

The training process of a prototypical network is called meta-learning since a prototypical network is not optimized for a certain classification task with a fixed number of classes. It is rather optimized to perform better in a classification task where both class samples and query samples are given in the inference step. The encoder acquires a metric space (or how to vectorize) that the classifier achieves a better classification performance.

In the rest of this section, we describe how we design the encoder and classifier for our anomaly detection. We also explain how we train the prototypical network.

A. Encoder

Our encoder is an extended version of the code2vec model [14]. It takes the source text of a method body as its input and outputs a vector representation of that method body. The following describes the outline of our encoder's process:

1) Build an abstract syntax tree (AST) of given source text. For example, our encoder builds the tree on the left of Figure 2 from the source text shown on the bottom of the figure.

[2]https://github.com/zhuzhengping911/MySpringBoot
[3]https://spring.io
[4]https://mybatis.org

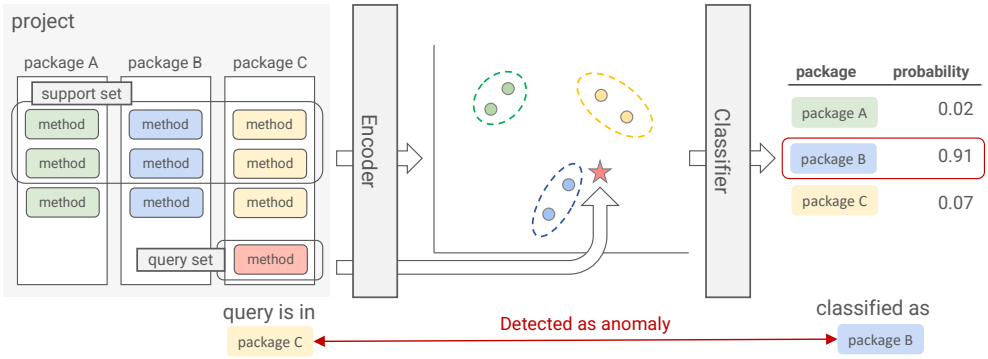
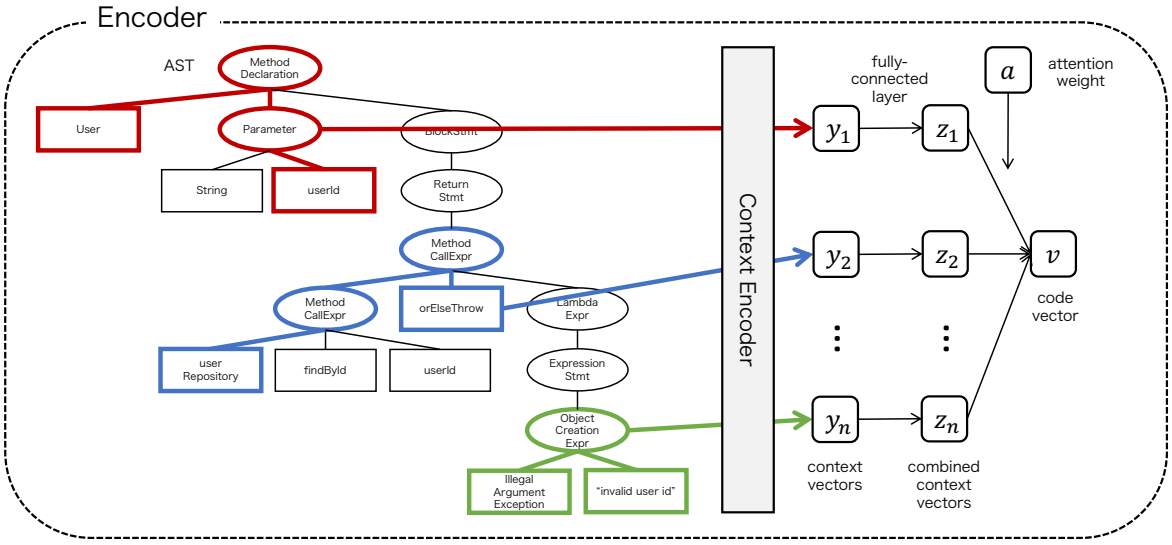


Figure 1: Overview of our detection system




```
public User getUser(String userId) {
  return userRepository.findById(userId)
    .orElseThrow(() -> new IllegalArgumentException("invalid user id"));
}
```

Figure 2: Code2vec encoder and example input

2) Randomly sample n pairs of leaf nodes in the AST and find the connecting path between nodes for each pair. The triplet of two leaf nodes and the path is called an *AST context*. In the AST of Figure 2, three AST contexts are illustrated with three colors. For instance, the nodes and path colored with red are the AST context from the `User` node to the `userId` node. We set n to 200 in our experiment shown in Section IV.
3) Vectorize each AST context with a neural model known as the context encoder [20]. Since the context encoder is a complicated module, we explain this step in more detail separately in the following paragraphs.
4) Combine the elements of each vector obtained in the previous step using a fully-connected layer. This step is formalized as follows:

$$\mathbf{z}_i = tanh(W_{in} \cdot \mathbf{y}_i),$$

where $\mathbf{z}_i$ is an output vector of this step, W_{in} is learnable parameters, and $\mathbf{y}_i$ is an output vector of the previous step.
5) Compute the weighted average of the vectors obtained in the previous step. The computation of the weighted average $\mathbf{v}$ is formalized as follows:

$$\mathbf{v} = \sum_{i=1}^{n} \left(\frac{\exp(\mathbf{z}_i^T \cdot \mathbf{a})}{\Sigma_{j=1}^{n} \exp(\mathbf{z}_j^T \cdot \mathbf{a})} \right) \cdot \mathbf{z}_i.$$

Here, $\mathbf{a}$ is learnable parameters, and $\mathbf{z}_i^T$ is the transpose of $\mathbf{z}_i$. This technique is known as the soft attention mechanism.

As we mentioned, the context encoder vectorizes an AST context. Let us denote an AST context as $(w_s, w_t, r_1 \ldots r_l)$, where w_s and w_t are the leaf nodes, and $r_1 \ldots r_l$ are intermediate nodes along the path. The context encoder (1)

vectorizes w_s, w_t, and $r_1 \dots r_l$, respectively, (2) concatenates the obtained vectors into a single vector, and (3) outputs the concatenated vector. To vectorize a node, the context encoder first splits the names in a node w into lower-case words. For example, if the target node is the `userId` node in Figure 2, it splits `userId` into `user` and `id`. The context encoder then vectorizes each word using an embedding matrix (a collection of word vectors) and dumps the average of obtained word vectors as the node vector. To vectorize a path, the context encoder first vectorizes every node on the path and put obtained vectors into a bi-directional LSTM [21], [22]. The output of the LSTM is the path vector. The vectorization of the context vector is formalized as follows:

$$\mathbf{y}_i = [enc\_token(w_s); enc\_path(r_1 \dots r_l); enc\_token(w_t)],$$

where $\mathbf{y}_i$ is the context encoder's output, enc_token is the node vectorizer, and enc_path is the path vectorizer. Here, $[\mathbf{v}; \mathbf{w}]$ represents the concatenation of two vectors $\mathbf{v}$ and $\mathbf{w}$.

Our encoder extension is in the context encoder. When it converts a leaf node into a set of words, we also add type information into that set. When a leaf node is an identifier, such as a variable name and a method name, the encoder obtains its static type T. If T is a class or interface type, the encoder also obtains its immediate superclass and interfaces. The names of all the obtained classes and interfaces are split into words as the variable and method names. Then these words are added to the set of words for that leaf node. Likewise, if a leaf node is a class or interface name, the encoder splits the names of its immediate superclass and interfaces into words and adds them to the set for the leaf node. Figure 3 illustrates our extension and how the context encoder works.

B. Classifier

Our classifier is the same as the classifier for the original prototypical network. It is not a neural network unlike other models for few-shot classification [23], [24]. This is an advantage of prototypical networks because the classifier can be easily constructed again without a time-consuming gradient descent algorithm when some methods are modified to or removed from the project.

The classifier for a project u is constructed with the support set for that project u. The support set is given as $D = \{(x_1, y_1), ..., (x_N, y_N)\}$, where x_i denotes the source code of a method m_i ($i = 1, 2, \dots, N$) and $y_i \in \mathbf{K}$ denotes the label of the package that includes that method m_i. Here, $\mathbf{K} = \{1, \dots, K\}$. For a package k, D_k denotes a subset of D where all the elements are the methods included in that package k. Then, the representative (central) point $\mathbf{c}_k$ for a package k is given as follows:

$$\mathbf{c}_k = \frac{1}{|D_k|} \sum_{(x_i, y_i) \in D_k} f_\phi(x_i) \tag{1}$$

$f_\phi(\cdot)$ is a function representing the encoder. When constructing the classifier, only $\mathbf{c}_k$ has to be computed for every k.

The outputs of the classifier are probabilities of plausibility for K packages. The probability for package $k \in \mathbf{K}$ can be computed as

$$p_\phi(y = k \mid q) = \frac{\exp\left(-dist(f_\phi(q), \mathbf{c}_k)\right)}{\sum_{k' \in \mathbf{K}} \exp\left(-dist(f_\phi(q), \mathbf{c}_{k'})\right)} \tag{2}$$

where q is the source code of a query method and $dist(\cdot)$ is the Euclidean distance function.

The classifier p_ϕ is used to determine whether a method m in the query set should be moved or not. Our technique predicts the package with the highest probability for that method m. Let $\hat{k} \in \mathbf{K}$ be the predicted package for m and $k \in \mathbf{K}$ denotes the current package of the method m. If both $k \neq \hat{k}$ and $p_\phi(y = \hat{k}) > \theta$ are fulfilled, our technique detects that the method m should be moved from k to $\hat{k}$, where $\theta \in [0, 1]$ is a threshold. We determine the value of θ in an empirical way. In our case study conducted in Section IV-B, we set θ so that the evaluation performance in section IV-A will be maximized.

C. Training encoder

We do meta-learning as the original proposal of prototypical networks did. We give a number of pairs of a support set and a query set so that the network will output more accurate probabilities for the query set than before. Note that only the encoder is a neural network and the classifier is not. The classifier is constructed for every support set. When we use the model after training it, we use the trained encoder as it is but we must reconstruct the classifier to fit the support set we give as an input with a query set.

To obtain the labels for supervised learning, we collected a number of Java projects from GitHub. Some methods in these projects might be placed in inappropriate packages. To mitigate their effects, we carefully selected only projects with many stars on GitHub. We assume that such projects should show good modularity. More details will be mentioned in Section IV.

Throughout the meta-learning, we train the learnable parameters ϕ for the encoder f_ϕ so that the classifier p_ϕ will show good performance. A single cycle of mini-batch training in the context of meta-learning is called an *episode*. For every episode, one project is chosen among all the projects in the training dataset. The probability of choosing a project $u \in U$ is $\frac{|D_u|}{\sum_{u' \in U} |D_{u'}|}$, where D_u denotes all the pairs of methods and its package included in the project u. Then K packages are randomly chosen in that chosen project and N methods are also randomly chosen from every package. The chosen $N \times K$ methods form a support set. Furthermore, some other methods are chosen from the K packages and form a query set. The support set and the query set are given to our model and the gradient descent algorithm is run to minimize a loss function $J(\phi) = -\log p_\phi(y = k \mid q)$. After iterating an episode many times, we obtain a well-trained encoder. This training is called N-shot K-way meta-learning.

D. Use case scenarios

A simple usage of our technique is to examine a new method committed to an existing project. After meta-learning,

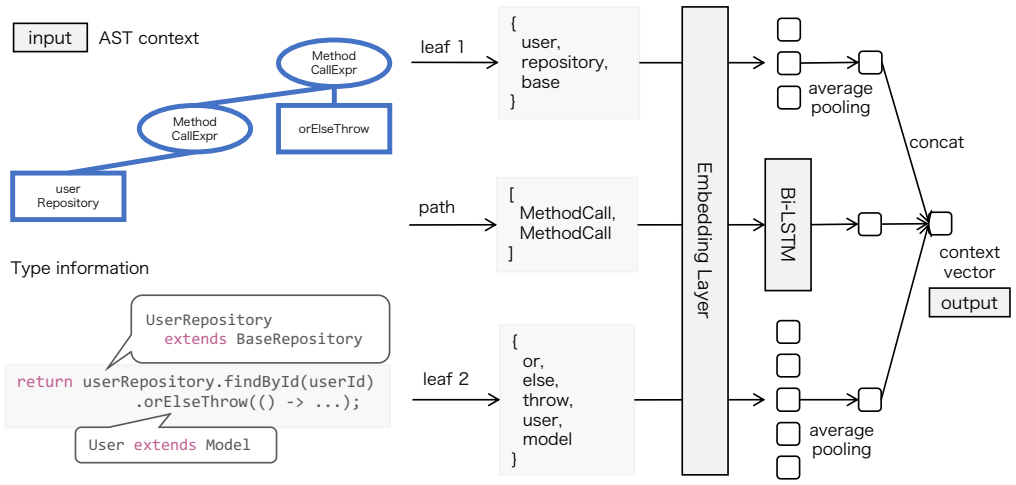


Figure 3: Our context encoder

TABLE I: Dataset

	training	validation	test	user-projects
projects	9005	250	300	1000
packages	2-2364	17-94	17-86	1-89
packages (avg.)	23.7	28.2	29.3	19.9
packages (std-dev.)	75.0	12.9	13.2	14.4

the classifier of our technique is constructed again by using all the existing methods in that project. Those methods are regarded as a support set for that project. Then the new method is examined by our technique to inspect whether the predicted package is the same as the current package.

Another scenario is to discover a method placed in an inappropriate package of an existing project. To do this, we divide all the methods in the project into a support set and a query set, for example, 80% for a support set and 20% for a query set. After the classifier is constructed again by using this support set, the query set is examined to find a misplaced method in that set. We iterate this by changing the support set and the query set so that all the methods will be included in the query set at least once.

IV. Evaluation

Our evaluation is three-fold and is described in the following three subsections. They correspond to these three research questions:

- **RQ 1:** How well can our technique detect anomalies in method placements?
- **RQ 2:** Are the detected anomalies indeed method misplacements in real-world Java projects?
- **RQ 3:** Are the design choices we make appropriate?

Dataset We adopt the *java-large* dataset that was originally for evaluating code2seq [20] and apply several modifications to make it suitable for our purpose. It contains more than 9,500 Java projects. We expect that the consistency of modularization in the projects included in this dataset is high because they are selected in the order of their star counts from GitHub.

We modify the java-large dataset to create our own dataset because some of the projects in the validation set and the test set have very few packages. Such projects are not suitable for evaluating whether our model can find anomalies. The modifications we make are as follows. First, we bundle all projects in the training, validation, and test set of java-large into one. Second, we randomly select 250 projects as the new validation set and 300 projects as the new test set among ones whose numbers of packages are between 15 and 100. Besides very small projects (less than 15 projects), we also exclude significantly large projects (100+ packages) so that the evaluation results are not blurred by exceptional cases. We use the remaining 9,005 projects as the new training set.

Table I shows several metrics of our dataset. Besides the projects adopted from the java-large dataset, our dataset also includes another set called *user-projects*. It consists of projects that we collect by ourselves from ones not in java-large. The details of the user-projects set are described in Section IV-B.

Preprocessing We preprocess our dataset in the following four ways. 1) Source code for automated testing are removed as much as possible because they are meant to be separated from other code in the same project regardless of the rules on modularity. We remove them by filtering out classes that have a subtoken `Test` in their names. 2) Abstract methods are excluded because they do not have method bodies to be input into our technique. 3) *Boilerplate code* are exclucuded because they tend to be scattered across packages regardless of the rules on modularity. We refer to getters, setters, overridden methods of `Object#equals`, `Object#hashcode`, and `Object#toString` as boilerplate. 4) All comments in the source code are removed.

We create word dictionaries for the use of word embedding in the neural network in the same way as Alon *et al.* [20]. We have two dictionaries: the collection of words that appear in the text of the methods and the collection of words that appear in the AST nodes. The word set from the method text consists of 1) tokens that represent reserved words,

2) subtokens derived from identifiers, and 3) subtokens derived from type information. We count occurrences of each word in the whole dataset and replace less frequent words with a special word <UNK>. This replacement is a typical way used in NLP (Natural Language Processing) tasks, which contributes to preventing neural-network-based models from overfitting.

Model Training For the meta-learning, we train our network for approximately 70,000 episodes. The training took 27 hours on an NVIDIA A6000 GPU. We set the learning rate to 0.001 and the meta-learning configuration to 10-shot 20-way. For each episode in the meta-learning, we select 5 samples for each package as the query set. Other hyperparameters can be found in the source code included in our artifact [16]. After the meta-learning, a classifier is constructed for each project to detect method misplacement that is specific to it.

Only 955 projects out of 9005 collected are used in the meta-learning due to a limitation of our training procedure. Projects that do not contain enough number of methods for an episode are skipped. For every episode, we need 200 methods for the 10-shot 20-way learning and additional $Q \times 20$ methods for the query set, where Q is a hyperparameter. We set Q to 5 as stated above, thus the total number of methods required for an episode is 300. Although the projects that are used in the meta-learning account only for 10.6% (= 955 / 9005) of the overall projects, they cover 72.7% (= 5,682,782 / 7,821,121) of the overall methods.

A. Anomaly Detection Performance

Evaluation Method To answer RQ 1, we examine the anomaly detection performance of our technique. We also compare the result to another tool that we create by adapting an existing learning-based model to our purpose.

The detection performance is evaluated by their *accuracy* and *F1 score*. They are calculated as follows:

$$precision = \frac{\text{\# of true positives}}{\text{\# of true positives+\# of false negatives}}$$
$$recall = \frac{\text{\# of true positives}}{\text{\# of positive samples in the dataset}}$$
$$F1\ score = \frac{2 \cdot precision \cdot recall}{precision + recall}$$
$$accuracy = \frac{\text{\# of correct recommendations for true positives}}{\text{\# of true positives}}$$

The precision measures how well our system can find anomalies in method placements. For example, if a method is misplaced and our system outputs a package other than the one that it currently belongs to, that is a true positive. The accuracy measures how well our system can recommend a package that a method should be moved to. The *correct recommendation for true positives* is the number of true positives whose destination packages match the ground truth.

To know the ground truth of method placements, we intentionally misplace methods in the test set by moving some methods in a project to different packages. Our intentional misplacements mimic ones that may appear in the wild. For every method in a project, we decide if it is moved with 1% probability. For method m, the move destination is selected randomly from packages described below:

1) packages that include methods calling m,
2) packages that include methods called by m, and
3) packages that have a same import statement as one of the import statements in the class file where m exists.

In addition to evaluating our technique alone, we compare it to another tool that we create by adapting an existing learning-based method to our purpose. The goal of this comparison is to show that straight-forward modification of existing methods is not enough to achieve high detection performance. To this end, we use the model by Liu *et al.* [15] that is the state-of-the-art as of writing in detecting *feature envy* [18] and outperforms well-known JDeodrant. Feature envy is a method that references variables and methods in other classes more often than ones included in its own class. We say that this method *envies* other classes rather than its own. We modify Liu's method so that it detects methods that envy other packages (not methods) rather than their own.

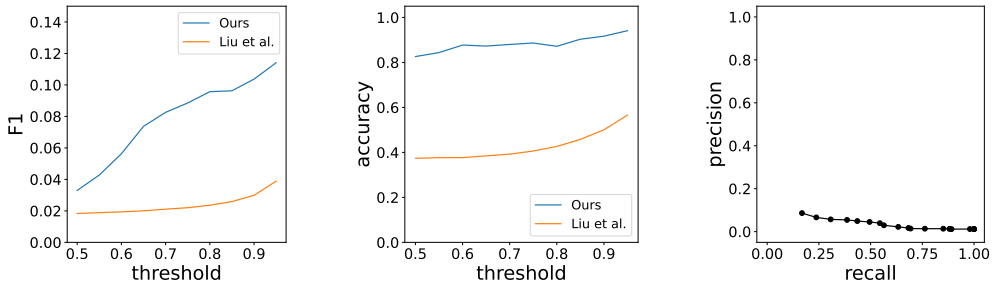
Liu's model is modified in the following four ways:

1) The input is changed from class-wise metrics to package-wise metrics. While the original method receives the name of a class and the distance between the class and a method, our modified version receives the name of a package and the distance between the package and a method. A distance is calculated in the same way as in [15] except that a class is replaced by a package.
2) Liu's model with the input changed takes a method m, the package k_1 that m belongs to, and another package k_2 as the input and outputs the probability that m should be moved from k_1 to k_2. On the other hand, our model directly outputs a package k that m should be moved to and its probability. Thus, we input all candidate packages for m to modified version of Liu's model and regard the package with the highest probability as the one that m should be moved to.
3) A probability threshold is introduced. If the highest probability that Liu's model outputs for a given method is smaller than the threshold, we regard the model predicts that the method should not be moved.
4) The model is trained on our own dataset to enable fair comparison.

Result The F1 score and the accuracy with different threshold values are shown in Figure 4 (a) and Figure 4 (b), respectively. The following paragraphs describe key takeaways we draw from the result.

Takeaway 1: Our technique achieved very high accuracy in any threshold value. The highest accuracy was as high as 0.941 with the threshold value of 0.95. This means that the packages that our technique predicted as the move destination matched the ground truth in 94.1% of the cases. The result shows the effectiveness of our model design and the use of few-show learning for anomaly detection in method placements. To clarify how our design choices affect the results, we conduct an ablation study in Section IV-C.

Takeaway 2: Our technique achieved significantly higher performance than the modified version of Liu's model. The



(a) F1 Score (b) Accuracy (c) PR curve

Figure 4: Evaluation result of detection performance

highest accuracy that Liu's model achieved was 0.566 with the threshold value of 0.95, which is almost 40 points lower than ours. Note that this result is different from the detection accuracy of feature envy shown in [15] because our evaluation (1) requires to find broader range of anomalies in method placements than feature envy, and (2) includes more move candidates (all packages in a project) to choose from than the experiments done in [15] (ones detected as move-refactoring-capable by Eclipse JDT [25]). The result shows that straightforward modification of an existing learning-based method is not enough to detect anomalies in method placements.

In terms of the F1 score, it is less than 0.115 for our model even in the best case, although it is slightly higher than Liu's model. To understand this deeply, we show the PR (Precision-Recall) curve in Figure 4 (c). The precision is low everywhere while the recall is high at certain threshold values. This means that our technique mistakenly suggests that a method should be moved in many cases even if it should not be moved indeed. Among these false positives, ones that move methods between two distinct projects can be filtered out when our technique is used in actual software development. These suggestions happen in our evaluation because a GitHub repository can include more than one project, but we do not distinguish them when creating the ground truth. For example, application code and library code that are independently developed are two distinct projects, but they may be bundled together in one repository. Our technique often suggests that a method be moved between them because they implement similar functionalities. However, developers can choose not to feed distinct projects at once to our technique in actual software development.

B. Case Study

Evaluation Method To answer RQ 2, we conduct a case study that applies our technique to real-world Java projects. We categorize *valid* anomalies among detected ones either to actual method misplacements or not by manual investigation. Valid anomalies are ones that are neither *suspended* nor *excluded*. Suspended and excluded anomalies are ones that we cannot confirm as method misplacements or not due to our lack of project-specific knowledge, and that suggest methods be moved between two distinct projects, respectively. As discussed in Section IV-A, excluded anomalies can be avoided in real software development by not feeding distinct projects together to our technique.

A valid anomaly is regarded as a method misplacement if it satisfies either of the following conditions:

1) it violates general modularity principles (e.g., low-coupling and high-cohesion [2], information hiding [3]),
2) it violates modularization rules of the adopted framework (e.g., Model-View-Controller architecture), or
3) it violates project-specific rules on modularity, which are extracted from each project by manually reviewing the source code.

We create a new set of projects called user-project for this study. Unlike the java-large dataset, this set only contains projects owned by *personal accounts*[5] on GitHub. We collect projects of personal accounts because they tend to be developed by a small number of people and thus may have more anomalies in method placements than ones of organization accounts. This way we mimic a situation where a certain number of anomalies exist but have not yet been fixed. Among the most-starred projects owned by personal accounts, we collect ones that are created within the last 10 years and not included in the java-large dataset. We collect more than 7,000 projects and randomly pick 1,000 projects from them to be used in the case study.

Result Figure 5 shows the breakdown of the valid anomalies. We detected 319 anomalies in total, out of which 171 were valid, 40 were suspended, and 108 were excluded. The detection threshold of the model was set to 0.95, which achieved the highest F1 score in Section IV-A. We draw two takeaways from the result.

Takeaway 3: 51 valid anomalies out of 171 (30 %) were actual misplacements. This shows that our idea of detecting anomalies in method misplacements is beneficial for finding modularity rule volations. The reason why the result is better than the precision in Section IV-A (see Figure 4 (c)) can be because we excluded cross-project suggestions in this section as developers would do in actual software development.

Takeaway 4: The suggested move destination was correct

[5]https://docs.github.com/en/get-started/learning-about-github/types-of-github-accounts#personal-accounts

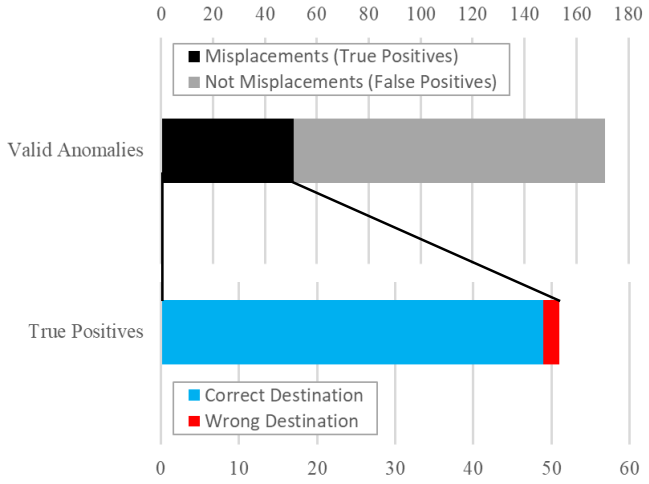


Figure 5: Breakdown of all valid anomalies

in 49 true positives out of 51 (96 %). This confirms that our model design and the use of few-shot learning are effective for actual method misplacements, and our technique can be leveraged in real-world sotftware development.

Analysis of True Positives We analyze the true positives to clarify that our technique indeed detects actual misplacements that stem from modularization rule violations. Listing 1, 2, Listing 3, Listing 4, and Listing 5 are example true positives. The following paragraphs explain why they are regarded as modularization rule violations that may damage the code quality and maintanability of the project. An advantage of our technique is that it detects all of the below examples without explicit knowledge on which modularization principle is used in each project. On the otherhand, using existing techniques such as measuring code quality metrics and applying a learning-based model for a specific type of misplacement (e.g., feature envy) to find them requires human involvement on which metric to measure and which model to apply. These are not always easy tasks especially in the later stage of software development.

Listing 1 contains a method that implements a different concern than the one of its containing class and should be moved to the class in Listing 2, and Listing 3 violates the principle of separation of interfaces and their implementation. Detailed descriptions of these code are in Section II.

Listing 4 is an example of *brain controller* [26] and violates the modularization rules of the underlying framework. Our technique suggested that this method should be moved from the `controller` package to the `service` package. It is a part of a web application[6] that adopts SpringMVC as its framework and violates the rule that a controller should not contain business logic. This method contains a business logic that checks the validity of a user's password and saves it, which should be done in the `service` package according to the standard modularization rule of SpringMVC.

[6]https://github.com/6324/xxl-job-admin-postgre

```
package controller;
public class UserController {
  @RequestMapping("/update")
  @ResponseBody
  @PermissionLimit(adminuser = true)
  public ReturnT<String> update(
      HttpServletRequest request,
      XxlJobUser xxlJobUser) {
    XxlJobUser loginUser = (XxlJobUser) request
      .getAttribute(LoginService.LOGIN_IDENTITY_KEY);
    if (loginUser.getUsername()
      .equals(xxlJobUser.getUsername())) {
      return new ReturnT<String>(
          ReturnT.FAIL.getCode(),
          I18nUtil.getString(
              "user_update_loginuser_limit"));
    }

    if (StringUtils.hasText(
      xxlJobUser.getPassword())) {
      xxlJobUser.setPassword(xxlJobUser
        .getPassword().trim());
      if (!(xxlJobUser.getPassword().length()>=4 &&
        xxlJobUser.getPassword().length()<=20)) {
        return new ReturnT<String>(
            ReturnT.FAIL_CODE,
            I18nUtil.getString(
                "system_lengh_limit")+"[4-20]");
      }
      xxlJobUser.setPassword(
          DigestUtils.md5DigestAsHex(
              xxlJobUser.getPassword().getBytes()));
    } else {
      xxlJobUser.setPassword(null);
    }

    xxlJobUserDao.update(xxlJobUser);
    return ReturnT.SUCCESS;
  }
}
```

Listing 4: Controller contains business logic in its method

```
package minimalpoem;
public class MainApp {
  private DaoSession initGreenDao() {
    DaoMaster.DevOpenHelper helper = new DaoMaster
        .DevOpenHelper(this, config.getDBName());
    SQLiteDatabase sqLiteDatabase = helper
        .getWritableDatabase();
    DaoMaster daoMaster =
        new DaoMaster(sqLiteDatabase);
    return daoMaster.newSession();
  }
}
```

Listing 5: A method that envies another package than its own

Listing 5 is an example of methods that envy other packages than their own. It is a method placed in the entrypoint of an Android application[7] and our technique suggested that this method should be moved to the `minimalpoem.db` package. The purpose of this method is to perform the initialization process to use a database. This method obviously envies the `minimalpoem.db` package because it repeatedly references

[7]https://github.com/Deali-Axy/MinimalPoem

```
package rcvadapter;
public class RcvMultiAdapter<T> {
  public void addItemView(int viewType,
      RcvBaseItemView<T> itemView) {
    mItemViewManager.addItemView(viewType,
        itemView);
  }
}
```

Listing 6: False positive: Delegation pattern

the `DaoMaster` class included in `minimalpoem.db` but does not reference any other classes.

Note that a method in a true positive cannot necessarily be cut-and-pasted to the suggested package even if the suggestion is correct. This is because a method that implements two functionalities is suggested to be moved based on either of them. For example, the method in Listing 4 includes the `controller`'s responsibility (validating and storing a password) and the `service`'s one (returning the validation result). Therefore, refactoring this method requires decoupling the two responsibilities and moving lines of code corresponding to each of them to different packages. Among the 49 cases whose move destinations are correctly suggested, 24 cases (49%) fall into this category. Modifying our model so that it handles code with a finer granularity (e.g., a basic-block level) may enable it to suggest a smaller portion of code to be moved, but we leave it for future work.

Analysis of False Positives We also analyse the false positives to find some typical cases on which our technique does not work well.

The most common case is the *delegation pattern* [27]. Out of 120 false positives, 15 of them fall into this category. Listing 6 is an example. This is an excerpt from an Android application[8]. The `addItemView` method has no functionalities other than merely dispatching the parameters to a method of `mItemViewManager` that has the same name. Our technique mistakenly suggests that this method should be moved to the `rcvadapter.manager` package despite this method appropriately employs the delegation pattern. This wrong prediction comes from two characteristics of the delegation pattern. 1) A method body is often very short and has little information about its functionality. 2) The destination method of a delegation usually has the same signature as the origin method of the delegation.

The second most common case is where *dependency injection* is used under Spring Framework and 9 false positives fall into this category. Listing 7 is an example. This is an excerpt from a web backend application[9]. In Spring Framework, it is a common practice 1) to separate an instantiation of a class and its usage for utilizing the dependency inejection mechanism of the framework, and 2) to put the instantiation in a package called `config` as in this example. Our technique mistakenly

[8]https://github.com/vanish136/recyclerviewadapter

[9]https://github.com/huo785/springboot2.0-shiro-jwt-layui-thymeleaf-swagger-mybatis

```
package config;
public class RedisConfig {
  @Bean
  public RedisTemplate redisTemplate(
      RedisConnectionFactory connectionFactory) {
    RedisTemplate<String, Object> template =
        new RedisTemplate();
    ...
    return template;
  }
}

package service;
public class RedisService {
  @Autowired
  private RedisTemplate redisTemplate;

  public Boolean persist(String key) {
    if (null == key){
      return false;
    }
    return redisTemplate.persist(key);
  }
}
```

Listing 7: False positive: Separation of instance creation and its usage in Spring Framework

suggests that the `redisTemplate` method should be moved to the `service` package because the usage of the object created by `redisTemplate` is concentrated in the `service` package and also a `config` package tends to include miscellaneous methods in the projectts included in our dataset. Learning this pattern as an appropriate method placement is difficult for our technique because whether this pattern is applied or not relies only on a small annotation (`@Bean` in the code) and SpringFramework applies dependency injection behind the scenes.

C. Ablation Study

Evaluation Method We conduct an ablation study on our technique to answer RQ 3. This ablation study includes two perspectives. We verify 1) that the subtokenization and type utilization contribute to the anamaly detection performance and 2) that our encoder is the most suitable for our task among other possible neural-network-based methods.

For perspective (1), we implement two variants of our technique: *ours w/o types* and *ours w/o subtokenization*. They use the same encoder but do not use type information or subtoken information, respectively. Ours w/o types is constructed by not adding type information into a set of subtokens when encoding them. Ours w/o subtokenization is constructed by not dividing source code tokens into subtokens. The word dictionary is also created by not dividing tokens into subtokens.

For perspective (2), we use three encoders that are based on different neural network models. They embed the same method into different vectors as they utilize different kind of information extracted from the input method. The first one is the plain code2vec [14] model that takes an AST as its input. The second one is Bi-directional LSTM with soft attention [12] that takes a sequence of tokens as its input. The last one is

TABLE II: The result of the ablation study

Variants	Top-1 accuracy
Ours	**0.652**
w/o types	0.629
w/o subtokenization	0.605
Code2vec	0.594
GGNN	0.585
Bi-LSTM	0.314

GGNN [10] that takes a graph representing the method body as its input. To use GGNN with our technique, we attach a readout module adopted from [28] because the vanilla GGNN converts a single graph node into a vector but we need to convert a whole graph (i.e. a whole method body).

We evaluate each variant by their top-1 accuracy of method-to-package classification. This is because the anomaly detection performance of the whole implementation is dominated by this metric. The top-1 accuracy is the proportion of exact matches against all cases, where an exact match means when the actual package the query method belongs to is predicted with the highest probability by the classifier. Note that this terminology is different from the accuracy in Section IV-A. In this evaluation, we divide each project into five portions so that methods used for few-shot learning do not appear in the queries. The top-1 accuracy for a project is calculated by taking the average over the five portions.

Result Table II shows the top-1 accuracy of each variant. We draw two takeaways from the result.

Takeaway 5: The top-1 accuracy of "Ours" is higher than that of other encoders. This means that the design choices we make in Section III significantly improve the anomaly detection performance than using an existing encoder as-is. Compared to the best result besides "our", the difference is more than 8.8 % ($\frac{0.652-0.594}{0.652} = 0.0889...$).

Takeaway 6: The top-1 accuracy of "Ours" is higher than that of "w/o types" and "w/o subtokenization". This shows that combining type information and subtokenization togather largely contributes to improving the anomaly detection performance. Using only one of them does not rival the top-1 accuracy of "ours".

We expect that type information has the potential to improve the classification accuracy more than we observe in this study. This is because the type information we exploit is incomplete. Recent software projects often use various dependency management tools and it is sometimes time-consuming to understand how to download libraries and build projects. Due to this issue, we only make use of type information that can be extracted from the project source code itself, but not from libraries that it depends on. Feeding more complete type information to the model may result in better classification accuracy, but we leave it for future work.

V. Threats to Validity

Internal Validity The validity of our results in Section IV-B heavily depends on our manual inspection of the model outputs. We made guidelines to judge if the detected cases are truly misplaced methods or not, but the judgment we made can be still subjective. Furthermore, we are not aware of all the coding rules (especially undocumented ones) in the target projects since we are not the lead developers of those projects. To make our results publicly validatable, we published them as part of the artifact at Zenodo [16]. Specifically, we uploaded the spreadsheet file that contains the target code fragments, our system's suggestion and its probability, and our (manual) categorization.

External Validity The performance of our model depends on the quality of the dataset for model training. The performance may improve or deteriorate significantly if our model is trained with another dataset. As described at the beginning of Section IV, we carefully chose the dataset for our model based on the common assumption that most-starred repositories tend to be well-reviewed and thus well-modularized. Our model may achieve better performance if we train it with a collection of Java projects verified to be well-modularized.

VI. Related Work

Researchers have been developing techniques to detect misplaced code fragments. JDeodrant [17] and JMove [29] are well-known research artifacts for detecting code smells related to method misplacement (e.g., feature envy methods). Liu *et al.* presented a detection model for feature envy methods and reported that their model outperformed JDeodrant and JMove [15]. Palomba *et al.* proposed a technique to suggest the move class refactoring based on class cohesion metrics [30]. Hayashi *et al.* proposed an inference-based technique for a project in the MVC2 design [31]. The key difference between our technique and those existing techniques is that ours does not assume certain code smells (e.g., feature envy), project architecture (e.g., MVC2), or properties of misplacement (e.g., cohesion). Our model makes a judgment of whether a method is misplaced or not based on what it learned from ASTs and type information.

The study of misplacement detection is not confined to the detection of named misplacement patterns. Huynh *et al.* proposed a detection technique that uses design structure matrices [32]. Wong *et al.* presented a detection technique that uses co-change patterns among files in version history as well as dependencies among modules [33]. Some of architectural smells [4] are about code misplacement. Cunha *et al.* presented a technique to detect two architectural smells known as unstable dependency and god Component using machine learning [34]. Díaz-Pace *et al.* proposed a machine-learning technique to find cyclic dependencies and hub-like dependencies [35]. The study of architectural smell detection is not limited to the learning-based approach. Mo *et al.* formalized of several architectural smells and developed the detector of those smells. However, since these techniques use architecture-level features to detect misplacements, they cannot suggest exactly which method causes modularity violations. Our technique, on the other hand, can suggest to which package a misplaced method should be moved.

The application of few-shot classification techniques in the software engineering field is not well discovered, whereas the techniques are widely used in the field of image classification and natural language processing tasks [36], [37]. The paper [38] mentions the use of few-shot learning techniques for the program synthesis and presents a dataset for it. However, it does not propose an actual use of the dataset. To the best of our knowledge, this study is the first application of a few-shot classification technique for a code-related task.

VII. Conclusion

In this paper, we presented a technique to detect anomalies in method placement in a Java project. Our technique can find project-specific anomalies, not only generic or project-agnostic anomalies, thanks to the few-shot classification technique. Our empirical evaluation revealed that our technique outperformed an existing feature-envy detection model. It also revealed that 30% of detected anomalies were misplaced methods and that suggested move destination was correct in 96% cases. We also conducted a case study and ablation study to validate the output and design of our neural network model.

References

[1] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[2] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, 1974.

[3] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, 1972.

[4] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009.

[5] A. L. Wolf, C. Fl, D. Perry, D. E. Perry, and E. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, 1992.

[6] L. Hochstein and M. Lindvall, "Combating architectural degeneration: a survey," *Information and Software Technology*, vol. 47, no. 10, pp. 643–656, Jul 2005.

[7] F. Beck and S. Diehl, "On the congruence of modularity and code coupling," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[8] H. Abdeen, S. Ducasse, and H. Sahraoui, "Modularization metrics: Assessing package organization in legacy large object-oriented software," in *2011 18th Working Conference on Reverse Engineering*, 2011.

[9] J. Snell, K. Swersky, and R. Zemel, "Prototypical networks for few-shot learning," in *Advances in Neural Information Processing Systems*, 2017.

[10] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018.

[11] Y. Wang, H. Yu, Z. Zhu, W. Zhang, and Y. Zhao, "Automatic software refactoring via weighted clustering in method-level networks," *IEEE Transactions on Software Engineering*, 2018.

[12] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016.

[13] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*, 2018.

[14] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, 2019.

[15] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[16] Anonymous, "Research artifacts - an anomaly-based approach for detecting modularity violations on method placement," Mar 2022.

[17] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011.

[18] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[19] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, 2009.

[20] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2019.

[21] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[22] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, 1997.

[23] O. Vinyals, C. Blundell, T. Lillicrap, k. kavukcuoglu, and D. Wierstra, "Matching networks for one shot learning," in *Advances in Neural Information Processing Systems*, 2016.

[24] F. Sung, Y. Yang, L. Zhang, T. Xiang, P. H. Torr, and T. M. Hospedales, "Learning to compare: Relation network for few-shot learning," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

[25] Eclipse Foundation, "Eclipse Java development tools (JDT)," https://www.eclipse.org/jdt/.

[26] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. Deursen, "Code smells for model-view-controller architectures," *Empirical Software Engineering*, 2018.

[27] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[28] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[29] R. Terra, M. T. Valente, S. Miranda, and V. Sales, "Jmove: A novel heuristic and tool to detect move method refactoring opportunities," *Journal of Systems and Software*, 2018.

[30] F. Palomba, M. Tufano, G. Bavota, R. Oliveto, A. Marcus, D. Poshyvanyk, and A. De Lucia, "Extract package refactoring in aries," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.

[31] S. Hayashi, F. Minami, and M. Saeki, "Inference-based detection of architectural violations in mvc2," in *Proceedings of the 12th International Conference on Software Technologies - Volume 1: ICSOFT,*, 2017.

[32] S. Huynh, Y. Cai, Y. Song, and K. Sullivan, "Automatic modularity conformance checking," in *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008.

[33] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011.

[34] W. S. Cunha, G. A. Armijo, and V. V. de Camargo, "Inset: A tool to identify architecture smells using machine learning," in *Proceedings of the 34th Brazilian Symposium on Software Engineering*, 2020.

[35] J. A. Díaz-Pace, A. Tommasel, and D. Godoy, "Towards anticipation of architectural smells using link prediction techniques," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2018.

[36] Y. Bao, M. Wu, S. Chang, and R. Barzilay, "Few-shot text classification with distributional signatures," in *International Conference on Learning Representations*, 2020.

[37] M. Yu, X. Guo, J. Yi, S. Chang, S. Potdar, Y. Cheng, G. Tesauro, H. Wang, and B. Zhou, "Diverse few-shot text classification with multiple metrics," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, 2018.

[38] F. Alet, J. Lopez-Contreras, J. Koppel, M. Nye, A. Solar-Lezama, T. Lozano-Perez, L. Kaelbling, and J. Tenenbaum, "A large-scale benchmark for few-shot program induction and synthesis," in *Proceedings of the 38th International Conference on Machine Learning*, 2021.