Department of Creative Informatics
Graduate School of Information Science and Technology
THE UNIVERSITY OF TOKYO

Master's Thesis

A Detection System of Inappropriate Method Placements in Java Projects

Java プロジェクトにおけるメソッド配置の誤り検知システムの開発

Kazuki Yoda
依田 和樹

Supervisor: Professor Shigeru Chiba

January 2022

Abstract

It is important to detect and fix code that badly affects the architecture design of the project in order to keep developer productivity at a high level. Code smell is often referred to as an indicator of design decay and a number of studies have been made to automatically detect them. Most studies have utilized structure-based metrics to detect them. These structure-based methods have achieved good performance in detecting code smells, but it is claimed by an empirical study that not a few numbers of detected cases are not very relevant to architectural design problems. Besides, the empirical study also shows that most architecturally-relevant bad code can be seen in modules that have mixed concerns. In this study, we propose and implement a system that detects methods placed in inappropriate packages targeted at Java systems. They can be regarded as inter-module misplacements and have a negative influence on their architecture designs by increasing coupling between modules. To detect them based not on structural features but on concerns, we exploit a neural network model to extract semantic features of source code. Moreover, we utilize few-shot classification using prototypical networks as an internal structure of our detector in order to consider the project-specific differences in design rules. We evaluated our model by comparing detection performance to a similar model as ours that also uses a neural network, and showed that our model outperformed it. We also conducted a case study targeted at real-world software projects and successfully detected cases that badly affect architecture designs.

概要

アーキテクチャデザインに悪影響を与えるコードを検出・修正することは開発者の生産性を高く保つ上で重要である．コードスメルはそれらの兆候として扱われ，それらを自動的に検出するための手法が提案されてきた．多くの手法では主にソースコードの構造に基づくメトリクスを用いた検出が行われてきた．これらの構造に基づく手法では高い精度でコードスメルを検出したが，同時にその事例の多くが実際にはアーキテクチャデザイン上の問題とは関連が薄いことも実証研究によって示されてきた．また，同研究によってアーキテクチャデザインに悪影響を与えるコードは関心が混在したモジュールに多く見られることも指摘されている．そこで，本研究では Java システムを対象に，配置するパッケージを誤ったメソッドを検出するシステムを提案・実装した．このようなメソッドはモジュールを跨いで配置を誤っており，複数のモジュール間の結合度を高めるなど，アーキテクチャに悪影響を与える可能性が高いと考えられる．我々は，コードの構造ではなく関心に基づいた基準でそれらを検出するため，ニューラルネットワークを用いてコードの意味的な特徴を抽出するようなモデルを用いた．また，検出器の内部構造として prototypical networks を用いた few-shot 分類を行うことで，プロジェクトごとのデザインルールの違いを反映した検出ができるようにした．実装したモデルの評価として，同様にニューラルネットワークを用いる既存手法との比較を行い，その検出性能を上回ることを確認した．また，現実のプロジェクトを対象としたケーススタディを行い，実際にアーキテクチャデザインに悪影響を与えるコードを検出できることを示した．

Contents

Chapter 1

Introduction

1.1 Importance of Code Quality

It has been widely accepted that writing good code is important. When writing code, we have to follow coding conventions, architecture designs that a project adopts, modularity principles, and so on. Following these rules is said to contribute to improving developers' productivity. On the other hand, if violated, that negatively affects the system's understandability, testability, extensibility, and reusability [1]. There are not only prevailing principles but also detailed project-specific coding rules we have to follow. Therefore, even experienced developers sometimes violate these rules if they are not very careful. To prevent violating rules, we usually detect them through peer code review. Developers mutually review other developers' code to confirm there are no problems in the code differences before accepting them.

Software assistant for code review is a significant task since it is a costly task. Developers have to spend their time on code review besides their main tasks. Due to this, a number of tools have been developed in order to assist code review. Code linter is an easy-to-automate example. It detects and fixes source code that does not follow project-defined format rules and it can be easily achieved by converting a parse tree into code using given format rules. By contrast, design-related problems are examples that cannot be easily detected and fixed. This is because architectural design rules are the underlying structure of software and they cannot be represented on the surface of source code. This is also a difficult task for humans. Hence code review is a difficult and time-consuming task. If we detect and fix these problems automatically, we can dramatically reduce the time for code review and keep code clean and productive. Therefore, tasks of detecting or fixing modularity problems have been one of the primary concerns in the software engineering field.

1.2 Previous Studies in Software Modularity

Software architecture is usually represented as a set of modules. Hence, maintaining good modularity is important in making software projects well-architected.

However, it is usually a difficult task to automatically detect bad code from the viewpoint of software modularity. This is because bad cases have so large varieties that we cannot patternize all of them. Nevertheless, we can partially solve the problem if we know tangible features of specific bad code patterns.

Previous studies have made a lot of efforts on patternizing and detecting bad code. Bad code has been patternized as code smell [2], a catalog of bad patterns. On the other hand, good modularity principles have been described by previous studies. Based on these modularity principles or bad code patterns, a number of studies have been tried to obtain

quantitative features of bad code.

1.2.1 Modularity Principles

Separation of Concerns

Separation of concerns [3] can be the most well-known principle in software development. According to this principle, we can improve software qualities in adaptability, maintainability, extensibility, and reusability by separating different concerns into distinct sections. For example, in the development of a client-server application, the system is typically separated into three layers: presentation layer, application layer, and data layer. Each layer has its own responsibility. By separating concerns, developers can modify each layer independently from other layers and it contributes to the team's productivity.

Information Hiding

Information hiding [4] is a modularity principle that encourages modules to hide design decisions that are likely to change inside them and to provide slim interfaces to other modules. This principle is useful to limit the software changes within the module boundaries. Following information hiding, when the software changes, the change will not affect other modules. This principle is also implemented as encapsulation in object-oriented programming.

Low Coupling and High Cohesion

Low coupling and high cohesion principle [5] is a principle for data communications in each module. They claim that good modularity characteristics are 1) that the number of intra-module dependency connections is high and 2) the number of inter-module dependency connections is low.

1.2.2 Code Smell

Fowler and Beck introduced a concept of *code smell* [2], that references bad code patterns, especially in object-oriented programming. Code smell is a set of bad code patterns. In addition, some of them are not only bad code but also can be indicators of modularity degradation. We introduce two commonly mentioned kinds of code smell in this section.

God Class

God class is a class that has references to a large number of other classes and/or has so many unrelated methods in it. God class implements all the functions in a single class that should be realized by multiple classes, and it takes on a great deal of responsibility. This is why such a class is called *god* class. God classes violate separation of concerns at this point. God class is a seemingly useful class because it implements many kinds of functions. However, its usefulness also means the class has so many couplings between other classes since most of the other classes have to use the god class. Moreover, the code lines of the god class are basically very large. When a developer modifies a method in a god class, checking the scope of the modification requires looking at the entire long class. Long code makes the maintenance difficult and sometimes can be a cause of embedding bugs.

Feature Envy

Feature envy is described as a method or an attribute that is "more interested in a class other than the one it actually is in" [2]. The example is shown in Figure 1.1. In the left hand side of Figure 1.1, `envyMethod` has 4 references to `ClassB` despite it has only

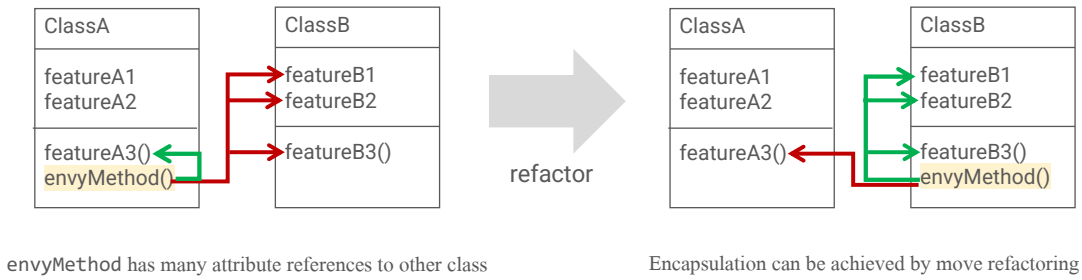


Fig. 1.1. The example of feature envy

1 intra-class reference. Hence, `envyMethod` is concluded that it envies the features in `ClassB`. This method violates low coupling and high cohesion principle. This is because it increases the coupling between `ClassA` and `ClassB` and lower the cohesion of each class. We can solve this problem by move method refactoring. In the right-hand side of Figure 1.1, `envyMethod` had been moved to `ClassB`, and we can see the problem no longer exists in this figure. Note that this problem cannot simply be described by the count of attribute references. Beck and Fowler used the word "more interested" in their definition, which means feature envy is a problem that stems from not only attribute references but also interests.

1.2.3 Existing Methods

A number of studies have tried to detect code smell. One of the typical ways of detecting code smell is to use code metrics. Code metrics are features of source code defined in mathematical ways based on observations on bad code. An example of a code feature is a reference. For instance, if a method has more inter-class references than intra-class references, the method is suspected to be feature envy. Another example is textual information on the source code. Palomba *et al.* proposed a method to detect code smell based on textual similarity [6]. It is confirmed by an empirical study that some of these attempts have succeeded in code smell detection [7].

There are several attempts to solve problems in the software engineering field using neural networks in recent studies. Hu *et al.* used a deep neural network model for generating code comments out of source code [8]. Similarly, Iyer *et al.* proposed a neural model for summarizing source code as natural language sentences. In addition to them, Allamanis *et al.* and Wang *et al.* proposed methods of neural-model-based bug detection [9, 10]. These models successfully detected variable usages that have been mistaken for other variables. However, attempts for utilizing neural models for modularity problems are limited. We can find only a few studies in this field [11, 12].

1.3 Detection of Inappropriate Method Placements

In this paper, we propose a detection method of indicators of architectural design problems. Architectural design problems have a massive impact on a project. It sometimes results in large-scale refactoring across the entire project. Therefore, it is an important task to detect and solve these problems as early as possible. We consider our detection tool can be beneficial in the following points.

- We can save resources for code review.
- We can keep our code clean and productive.

- We can avoid disastrous refactoring in advance.

We address the detection task by developing a concern-based detector. This is because it is shown that existing methods that utilize structure-based metrics are not sufficient to detect really architecturally-relevant code. Although code smell is usually referred to as a key indicator of architectural design problems, an empirical study conducted by Macia *et al.* shows that code smells detected by existing metric-based methods are not really relevant to design problems [7]. This finding implies that simply detecting code smells cannot solve architectural design problems. This study also shows that many architecturally-relevant code smells are located in modules that realize multiple concerns. Therefore, we believe focusing on a module's concern is a promising way for detecting architecturally-problematic code.

We focus on inter-package misplacements of methods because they can be considered as indicators of architectural design problems. In this study, we reckon Java packages as architecture modules as previous researches did [13, 14]. We regard a method placement as inappropriate if a concern that a method implements is different from its enclosing package's concern or a part of methods that realizes a specific concern is scattered in other packages. These methods are problematic from the viewpoint of architectural design because they have a negative influence on their architecture designs by increasing coupling between packages.

Our detection model overperformed an existing similar model. While we developed a tool for detecting inappropriate method placements, this detection task can also be solved by an existing model developed by Liu *et al.* [11]. We conducted an experiment of comparing the performance of these two models and confirmed our model can achieve higher performance. The result shows that our model selection is more suitable for this detection task.

We utilize a neural network as our model to perform concern-based detection. Recent software engineering studies that exploit neural networks have revealed neural networks have strong potential in embedding semantic information such as concerns into vectors [15, 16, 11, 9]. We developed our model based on code2vec [16]. Although there are only a few studies that use neural networks for solving problems related to architectural design [11, 12], we believe these techniques can achieve promising performance in our task as well.

Our target task is method-to-package classification, *i.e.*, we train a neural-network-based model that classifies an input method into its plausible package. Our model uses data samples for each package to obtain package features before classification. After that, our model embeds an input method into a vector representation, measures distances between a method vector and package features, then calculates probabilities for each package based on the distances. Given a method in package X, if our model predicts that the method should be in package Y with a certain probability, we consider package X is inappropriate and its encompassing package should be Y.

We incorporate a technique of few-shot classification in our model to specialize a classification model in each target project. Few-shot classification is a problem of classifying an input into a correct class only with a few samples of target classes. We utilize prototypical networks [17] in our model, which is originally developed for few-shot image classification. Our model requires package features for every package in a project beforehand, which means we create a specialized classifier for each project. Although it enables us to consider the diversity of architectural design of each project in the classification, there is a problem of data amount. Generally, the number of data samples in a single project is not sufficient to train a neural network model. We deem few-shot classification techniques can mitigate this problem and contribute to the detection performance. These techniques have achieved promising results mainly in the image processing field. While there are

also some attempts in natural language processing tasks, this is the first attempt in a source-code-related task.

We evaluated our model from some viewpoints. We first compared detection performance to a similar model as ours that also uses a neural network, and showed that our model outperformed it. We also conducted a case study targeted at real-world software projects and successfully detected cases that badly affect architecture designs. Detected cases are architecturally-relevant bad code such as a one that fails to information hiding or a one that strengthens coupling between packages.

Our research contribution is two-fold:

- We propose a novel method to detect indicators of architectural design problems. Based on an assumption that inter-package method misplacements can be key indicators of architectural design problems, we developed a detection tool for them. Through the case study, we confirmed our model can really detect architecturally-relevant bad code out of reak-world software projects.
- We confirmed our model's performance is higher than an existing model. Although the detection of misplaced method can be approached by another similar model, our model can perform better than that. Our model is designed as a combination of code2vec-based model and prototypical networks. Our evaluation result shows that these component designs contributed to the detection performance.

1.4 Structure of This Thesis

The rest of this thesis is organized as follows:

Chapter 2: Inappropriate Method Placements
In chapter 2, we first introduce the concepts of architectural design problems in previous studies. Then we clarify that there are few studies that can detect architecturally-relevant bad code that cannot be detected without considering concerns.

Chapter 3: Detection System
In chapter 3, we describe our neural-network-based detection model and how to train it. We utilize a neural network as our detection model because recent neural network models can be expected to be able to capture concerns of methods effectively. Besides, we introduce a few-shot classification technique to train our model. Few-shot classification enables us to 1) utilize dataset effectively and 2) perform detection with specializing for a target project.

Chapter 4: Evaluation
In chapter 4, we evaluate our proposed method. We conducted three experiments in the evaluation: a comparison to a previous study, a case study that applies our model for real-world applications, and an ablation study. We show our model can really detect architecturally-relevant bad code by these experiments.

Chapter 5: Conclusion
Finally, chapter 5 concludes our research and indicates directions of future studies.

Our research artifacts are on our GitHub page[*1]. This is a private repository and requires permission from our laboratory to access the artifacts. The artifacts contain the following contents.

- our model and preprocessor's source code
- our dataset

[*1] `https://github.com/csg-tokyo/playground/tree/yoda/thesis`

- learned parameters of our model
- result of the case study

Chapter 2

Inappropriate Method Placements

The importance of maintaining software architectural design clean and well-modularized has been widely recognized in both academia and industry [18]. Architecture is the underlying structure of software. If architecture is designed in accordance with modularity principles such as low coupling and high cohesion [5], and information hiding [4], it contributes to developers' productivities. Actually, however, architecture design often suffers from its degradation as the system evolves by making design decisions that violate the initially-intended design. This is called an *architectural design problem*.

In this chapter, we first introduce the concepts of architectural design problems that have been studied in this field. After that, we present techniques and their metrics used to solve architectural design problems. By reviewing them, we make it clear what is covered and what is not covered in this research field. Finally, we present our target task as the conclusion of this chapter.

2.1 Concepts of Architectural Design Problem

2.1.1 Architectural Design Problems

Architectural design problems have been referred to by a number of studies. They explain design problems from various different aspects and they can be often seen in real-world software projects.

Architectural Erosion

Architectural erosion is the process of introducing bad code that violates the intended architecture design [19]. Therefore, this problem only happens where prescribed design specification exists.

Listing. 2.1 is an example of architectural erosion in a web application that adopts MVC2 (Model-View-Controller 2) architecture as its design specification. In MVC2 architecture, model's mutation (`task` is responsible for model here) is the responsibility of `model` module. In this example, however, `controller`'s code does mutate a model instance in the midst of three lines in the method. This code leads to an increase in coupling between modules and consequently the system goes more brittle. If there are other use cases that need to change task status to complete, they have to clone this code because this code is not declared as an independent method and therefore is not accessible from other classes.

Architectural Drift

Architectural drift, sometimes also referred to as *architectural anomaly*, is introduced by Perry and Wolf [19], the same authors who introduced architectural erosion. Architec-

```
package controller;

public class TaskController {
    ...
    public static void completeTask() {
        long id = Long.parseLong(params.get("id"));
        Task task = Task.findById(id);

        // these three lines are model's concern
        task.status = TaskStatus.COMPLETED;
        task.completedDate = new Date();
        task.save();

        render();
    }
    ...
}
```

Listing 2.1. An example of architectural erosion in MVC2 web application

tural drift occurs due to introducing poor architectural decisions that make existing architectural components inadaptable. Such design decisions obscure the responsibilities of modules and make developers easy to make mistakes in future design decisions. Note that architectural drift does not go against existing design principles, unlike architectural erosion.

The concrete example of architectural drift is shown as Listing 2.2 and 2.3. They are controllers in a web application that handles requests from external clients. Originally, the responsibility of this controller was only to call `musicService` (Listing 2.2). Afterward, for some reason, the system becomes to handle a video concern in addition to music one (Listing 2.3). This modification does not increase inter-module couplings but makes the controller's responsibility obscure, which is referred to as *ambiguous interfaces* [1].

```
public void handle() {
  musicService.call();
}
```

Listing 2.2. before modification

```
public void handle(
    String target) {
  if (target.equals("music")) {
    musicService.call();
  } else if (target.equals(
      "video")) {
    videoService.call();
  }
}
```

Listing 2.3. after modification

2.1.2 Patternization of Design Problems

There have been research activities for discovering typical patterns of architectural design problems. They have been sometimes reported as catalogs of bad patterns.

Architectural Smell

Garcia *et al.* proposed the first patternized catalog of *architectural bad smells* or simply *architectural smells* [1]. Architectural smell is described as "an architectural decision that

negatively impacts system quality.“ There were no generalized patterns of architectural design problems while researchers have recognized the existence of architectural design problems and how they badly affect software development. After several bad patterns were proposed, researchers tried to detect them [20, 21, 22, 12] and proposed other patterns of architectural smells. Azadi *et al.* summarized these patterns and made a catalog of architectural smells [23].

Unlike code smell, architectural smell is not a code-level smell but an architecture-level smell. Garcia *et al.* showed 4 patterns of architectural smells in their paper called *connector envy*, *scattered functionality*, *ambiguous interfaces*, and *extraneous connector*. We explain what does architecture-level mean with some examples of architectural smell.

Scattered Functionality Garcia *et al.* described scattered functionality as “a system where multiple components are responsible for realizing the same high-level concern and, additionally, some of those components are responsible for orthogonal concerns [1].“ Figure 2.1 is the illustration of scattered functionality. Modules A, B, and C are responsible for realizing a single concern named `Shared Concern`, while modules B and C are responsible for their own different concerns B and C. Based on this definition, we cannot recognize this problem is occurring only by looking for code lines or classes in a single module, but we can tell the symptom of scattered functionality by looking out over multiple modules. This is the reason why architectural smell is an architecture-level bad pattern.

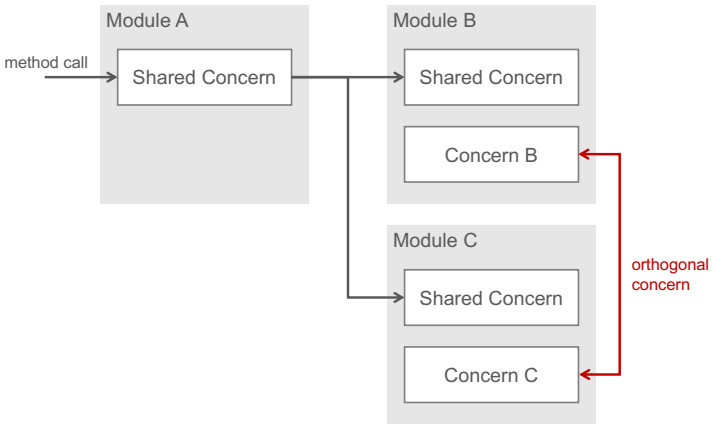
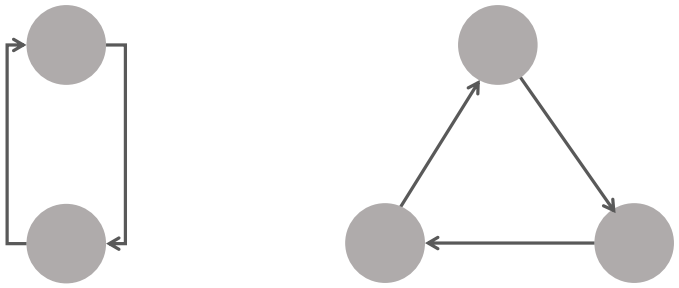


Fig. 2.1. Scattered Functionality

Cyclic Dependency Cyclic dependency [23] describes a system where its two or more modules depending on each module directly or indirectly (Figure 2.2). Dependency here means call dependency. The dependency graph of such a system has a circular part in it, and this is called cyclic dependency.

The cause of cyclic dependency cannot ask for lines of code or a single class. We have to look out over multiple modules to recognize cyclic dependency is occurring.

Speaking of patternization of architecture design problems, there is also a word *architectural antipattern* [24]. The difference between architectural smell and antipatterns is described by Garcia *et al.* [1]. They explain antipattern is the wider-range concept about software architecture, including project management and organizational problems other than architecture design itself. Therefore, they conclude that architectural smell is a narrower concept that only focuses on architecture design and its behavior.



(a) 2-modules cycle (b) 3-modules cycle

Fig. 2.2. Cyclic Dependency

2.1.3 Relevance to Code Smell

Code smell is often considered as an indicator of architectural design problems [2]. This is one of the major reason why a lot of researches have been conducted to detect code smells. Previous studies revealed that some specific types of code smell can be signs of architectural problems [20, 25].

Several studies also discuss code smells that can be design problems under specific architecture patterns. Aniche *et al.* [26] proposed code smell for Model-View-Controller architectures. They are bad patterns of implementation that can be often observed in MVC architectures and obscure the responsibility of each module of MVC architecture.

2.2 Metrics Utilization in Previous Studies

A number of studies have been conducted to detect code/architectural smells or to find refactoring opportunities. Most of these studies utilized source code metrics in their techniques, which can be divided into 2 major categories: structure-based metrics and concern-based metrics.

2.2.1 Structure-based Metrics

Structure-based metrics are so major metrics that most techniques utilize them to detect smells. Structure-based metrics can be computed from structural features extracted from source code like method calls. It also means structure-based metrics do not use textual information written in source code while concern-based metrics mainly use textual features.

Various kinds of structure-based metrics have been proposed so far. Well-known metrics suite are proposed by Chidamber and Kemerer [27] including LCOM (Lack of Cohesion in Methods) and CBO (Coupling Between Objects).

LCOM LCOM is represented as a 1-dimensional scalar number that stands for not-cohesiveness of a target class. The higher LCOM value is, the worse cohesion the class has.

This value is computed from the methods included in the target class and the set of instance variables referred to by the methods. Given a class C and set of its enclosing n methods $M_1, M_2, \ldots, M_n$, let $\{I_j\}$ denote a set of instance variables used by method M_i.

We can consider two types of sets:

$$P = \{(I_i, I_j) | I_i \cap I_j = \emptyset\}$$

$$Q = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}.$$

Then we can calculate LCOM as follows:

$$LCOM = \begin{cases} |P| - |Q|, & \text{if } |P| > |Q| \\ 0 & \text{otherwise} \end{cases}$$

This metric represents for not-cohesiveness of target class C.

We can guess the background hypothesis by this definition that a highly cohesive class has methods that refer to common instance variables. If there are two sets of methods that refer to respectively discrete two sets of instance variables, they should be split into two different classes.

This metric is categorized as a structure-based one. As we have been seen, LCOM uses instance variable uses as its basic feature. This is a feature that we can extract from the source code structure, not textual information. This is the reason why LCOM is called a kind of structure-based metric.

CBO CBO for a class C is defined as a count of the number of other classes that has a coupling to the class C. Coupling here means a method call or an instance variable references. If any of the methods encompassed by C has a method call or an attribute reference to another class X, then C and X are mutually coupled. This metric is also a structure-based one since its basic features are structural information, *i.e.*, inter-class references. According to low coupling and high cohesion principle, lower CBO is preferable.

These metrics have been widely used in this research field. Fontana *et al.* developed a tool named ARCAN [20]. ARCAN is an architectural smell detector and computes LCOM and CBO as one of the metrics used in a component. Other than detection tasks, they are also used for evaluating software modularity. Tufano used these metrics for analyzing when code smells are introduced by computing metrics for each commits [28]. Wang *et al.* proposed an automated refactoring tool that uses their own algorithms [10]. They used LCOM, CBO, and other metrics for evaluating the proposed refactorings really improve the cohesion and the coupling.

In addition to these well-known metrics, a variety of metrics have been proposed for each problem setting. Although such metrics are task-specific ones, most of them are computed from a few types of structural features:

- attribute references
- method calls
- class inheritance

Fokaefs *et al.* defined their own distance metrics between a method and a class using the number of attribute references to detect code smells [29]. Zanetti *et al.* also used attribute references to describe the relation between classes and modules as a graph [30]. They computed Newman's Q value as the cohesion and coupling metric of the target project in order to consider the way of move refactoring. Wong *et al.* used UML information in their study of modularity violation detection [31]. They computed their own metric called ACN (Augmented Constraint Network) using class inheritance and the relation between interfaces and their implementations on the class diagram.

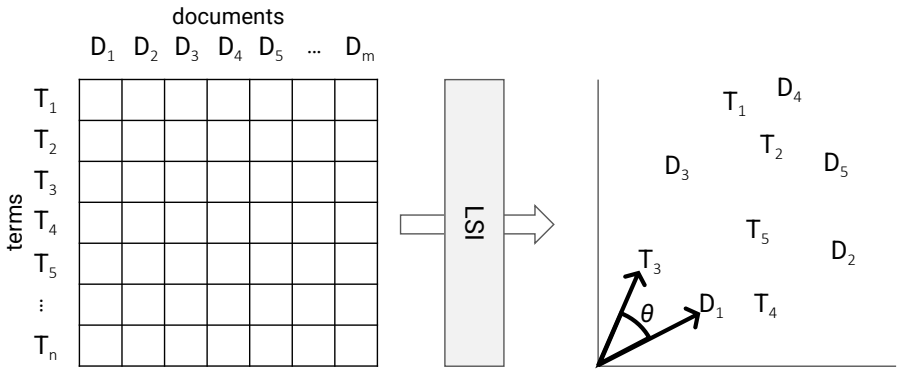


Fig. 2.3. Latent Semantic Indexing

2.2.2 Concern-based Metrics

Concern-based metrics are another interesting metrics although researchers have been paying less attention to them compared to structure-based metrics. We suppose this is because the concern is a difficult concept to be described in a mathematical way. All metrics we describe below utilize textual information of source code as a representation of concern. Therefore, the word *concern-based* here can be paraphrased with *textual-based*.

Textual information can be a good feature in capturing concern or semantic meaning of source code according to previous studies although the number of studies that use textual metrics is relatively smaller than structure-based ones.

LSI Kuhn *et al.* utilized LSI (Latent Semantic Indexing) in the semantic clustering of source code [32]. Palomba *et al.* also utilized LSI to extract textual feature of source code and exploit it for code smell detection [6]. LSI is originally developed in the language processing field and is used for the implementation of search engines' algorithms. They applied this technique for the source code task and showed that textual information of source code can be good features in extracting the semantic meaning of it.

LSI is a technique to transform documents and terms into latent vectors (Figure 2.3). Terms mean identifiers in source code and documents mean source files in code processing tasks. The input of LSI is the term-document matrix where each element a_{ij} represents how many times the term T_i appears in the document D_j. After the LSI algorithm is applied, we can get latent vectors of terms and documents. Then we can use these vectors to calculate the similarity between terms and documents by computing cosine similarities of the target pairs.

C3 C3 metric (Conceptual Cohesion of Classes) is proposed by Marcus *et al.* [33], which uses LSI vectors in its computation process. C3 metric represents concern-based class cohesion by 1-dimentional scalar value while LCOM also represents for not-cohesiveness of a class. Therefore, the usages of C3 are similar to the ones of LCOM but it is expected that C3 can capture more semantic cohesiveness.

When computing C3 metric, methods are regarded as documents and converted to document vectors using LSI. Then we can compute the conceptual similarity between methods (CSM) as the cosine similarity between method vectors. C3 value can be computed as the average of CSM values of all pairs of methods in a target class.

Bavota *et al.* used the combination of C3 and other structural metrics for decomposing refactoring of Java packages [34]. They also used C3 in another study that detects highly cohesive class groups out of existing modules and uses that information for extract class

```
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;
    }
    ...
}
```

Listing 2.4. The main activity class that has very detailed implementation

refactoring [35].

2.2.3 Other Approaches

There are some other approaches used to tackle architectural design problems. For example, Wong *et al.* used co-change patterns to detect modularity violations [31]. They use the revision history of the target project to mine co-change file clusters. This approach is neither structural nor textual.

Several studies that use neural networks have been proposed as well [11, 12]. The aforementioned metrics are all heuristic ones. Neural network-based approaches can be interpreted as the one that replace heuristics with approximated functions by training.

2.2.4 Limitation of Structure-based Metrics

Most studies about architectural design problems have utilized structure-based metrics and they have achieved promising results in their respective task settings. However, an empirical study conducted by Macia *et al.* [7] revealed an intriguing fact that most automatically-detected code smells are not relevant to architectural problems. They detected code smells using previously proposed techniques and examined that detected code smells are really relevant to architectural problems. As the result of their study, they found that more than 60% of detected code smells are not related to architectural problems. This result does not mean code smells have no relation to architectural smells but only trying to detect code smell cannot cover all indicators of architectural design problems.

2.3 Concern-based Detection of Inappropriate Method Placements

In this paper, we consider how we can detect methods placed in inappropriate packages. We consider Java packages as modules here as previous studies did [13, 14]. Inappropriate method placements can be interpreted as inter-module concern scattering, which is regarded as symptoms of architectural design problems.

```
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 2.5. Most of image processing functions are encompassed in this class

Listing 2.4 is an excerpt from a real-world application [*1] and an example of a method placed in an inappropriate package. This code is a part of the android application for photo edit. The `Editor` class seems the main activity of this application and its primary concern is to handle user actions on the device with managing states of the view. However, this class also has very detailed implementation such as `rotateImage` method as its private method. Hence, `Editor` class seems to be a kind of *god class* [2]. Ideally, the activity class should focus on handling user actions and delegate detailed tasks to other components that implement respective concerns. In addition, as we look out over the entire project, core functions of image processing are encompassed in `Utils` package (Listing 2.5). Therefore, we can conclude that `rotateImage` method should be moved to `Utils` package from the viewpoint of archutecture design.

We try to detect this kind of method in a concern-based way. Our motivation is based on the observation provided by Macia *et al.* that 64% of bad code related to architectural problems can be found in modules that have multiple concerns [7]. Besides, as we mentioned before, they also found that automatically-detected code smells using structure-based metrics are not really relevant to architectural problems. Furthermore, we can assume that developers recognize architectural problems grounded on a concern of each module according to the empirical study by Silva *et al.*, which uncovered that most developers perceive modules' cohesion based on their responsibilities. [36]. Therefore, concern-based ways can be considered as a promising way of detecting architecture-relevant bad code.

We present another example as Listing 2.6. This is also an excerpt from an existing software project on GitHub [*2]. The aim of this tiny project is a practice in developing web-based software in Java with the Spring framework[*3]. Thus the project adopts the Model-View-Controller (MVC) architecture. The method in Listing 2.6 is included in the `AuthorService_mybatis` class, a class classified into the model, which is called *service* in this project. In fact, this class belongs to the `service` package. MyBatis[*4] is a framework for database accesses. Since the `findAuthor` method is obviously a part of the service, it is declared in an appropriate place according to the MVC architecture. However,

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

```
package com.service; // -> com.service.impl

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

Listing 2.6. A concrete method in the `service` package

when we look at the `service` package, it seems that this package should include only the declarations of the interfaces to the *service* module. There is another package named `service.impl`, which includes the class implementing an interface in the `service` package. If this observation is true, the method declaration in Listing 2.6 is wrong with respect to the architecture. The `findAuthor` method should be changed into an `abstract` method, or an interface method, and its implementation should be moved in the `service.impl` package.

To point out such methods, we have to consider the correspondence between packages and concerns. In other words, we need to recognize the principles of what packages encompass what concerns. If we know the design rule that the image processing concern should be encompassed in the `Util` package in the example of Listing 2.4, we can claim that the image rotation method should be moved to there. Also about Listing 2.6, we can point out that the method is in the inappropriate package if we understand the difference of encompassing components between `service` and `service.impl`.

This is a difficult task to solve with existing known approaches because these methods do not consider package-concern correspondences. We have to look out for the way of recognizing the correspondence between packages and concerns in order to solve this task. The issue we tackle in generalizing this correspondence is that it varies depending on which framework the target project is adopting and on project-specific modularization rules. Narrowing down the detection scope only to web applications that adopt MVC2 framework, there is a similar work conducted by Hayashi *et al.* [37]. Although their work focuses on not methods but code lines and achieved good precision, it is the first attempt to focus on general architecture designs.

This work can be regarded as a variant of move method refactoring. A variety of techniques have been proposed in this field. However, there are no studies working on the same task as ours. Liu *et al.* worked on feature envy detection using structure-based metrics and deep neural network [11], which can be interpreted as detection of move method opportunities between classes. Although their work is similar to ours on the surface, they only target one type of code smell and do not consider modules and the relation between architectural problems. Bavota *et al.* proposed the tool named ARIES, which automatically decomposes concern-mixed packages by move refactoring [34]. Their purpose is sort of close to ours, but we focus on detecting each method as a move opportunity while their tool can only do package decomposition. Additionally, as far as we know, there are no prevailing studies that work on move method refactoring using concern-based ways.

Chapter 3

Detection System

3.1 System Overview

To point out inappropriate method placements based on a correspondence between a package and its encompassing concern, we develop a detector that captures method concern using a neural network. Previous studies have shown concern-based metrics that use textual information is useful to represent its semantic meanings [32, 6]. Likewise, recent neural-network-based techniques utilize textual information to consider concerns. In addition, they also utilize structural information such as AST structures, data flows, and control flows other than the text itself. They achieved excellent results in representing code semantics in these ways [15, 16, 11, 9].

Using neural networks enables us to automatically select important features out of source code. The difficulty in our problem is to determine what features of the source code contribute to the detection. This is mainly because 1) it is impossible to make general rules using structural features if we focus on concern-based detection, and 2) there so many types of misplacements that we cannot easily create manual rules. Therefore, using a neural network can be a suitable method for solving our problem since it can automatically select important features out of data samples.

On the other hand, in case adopting such a learning-based method, the learning process must be done for each project since design conventions are different respectively. Existing metric-based methods have formulated general rules and tried to apply them to software projects [20, 38, 30], which have achieved promising results. However, manually creating general rules for our problem is difficult because the naming convention and the way of dividing packages are totally different for each project, which means correspondence between concerns and packages of one project cannot be repurposed for others. Therefore we utilize a few-shot classification model that enables us to make different classifiers for each project. We describe the model details in the following sections.

Our target task is to detect methods that is placed in inappropriate packages and suggest the move refactoring of them to correct packages. The input of the system is the set of methods in the target project, and the output is the detection results and their correct packages. For example, we point out methods like Listing 2.4 and 2.6 and suggest Listing 2.4 should be moved to `Utils` package and Listing 2.6 should be moved to `service.impl` package.

There are two issues in solving this task using a neural network. The first issue is how we collect ground-truth data. Our task is classifying methods into their correct packages but there are no labeled data of method-package pairs. We collect well-modularized OSS (Open Source Software) projects from GitHub and utilize their method-package correspondence as the labels of supervised learning because it is impossible to manually annotate a huge amount of data. This is a kind of weakly-supervised learning since there

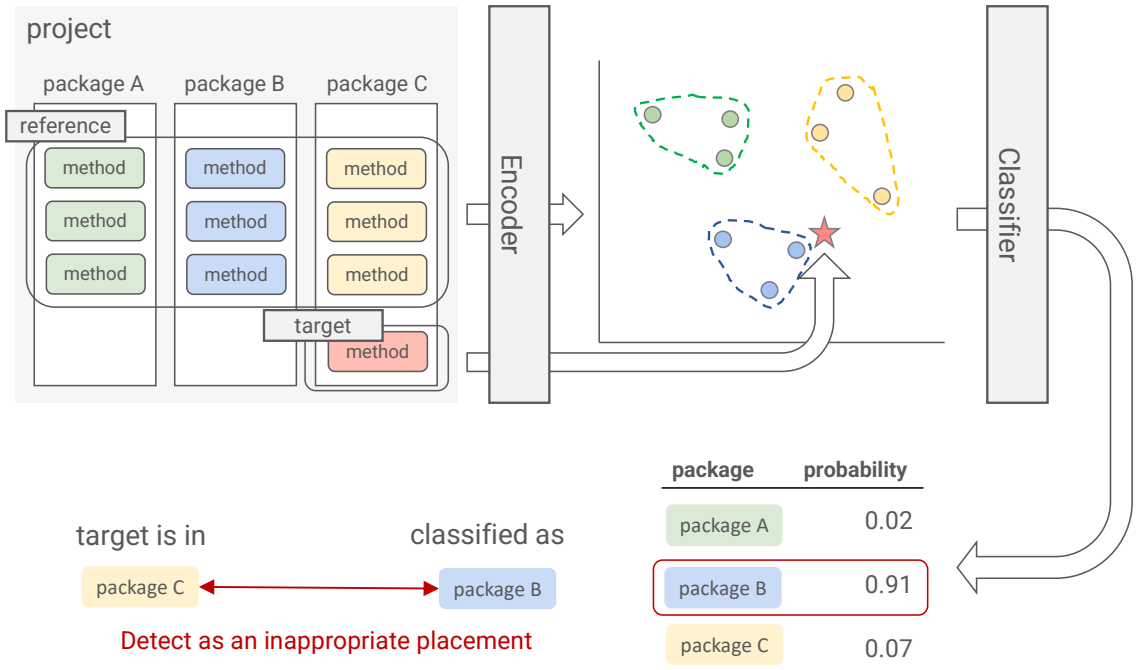


Fig. 3.1. Overview of our detection system

might be some noisy samples in the dataset. We acknowledge these noises and try to mitigate them by filtering projects in the dataset. Our dataset consists of the projects that have many stars in GitHub because such projects should have better modularity than other projects that have fewer stars. A detailed description of our dataset is written in section 4.1.

The second issue is how we train our model using the dataset collected from GitHub. Our dataset is a set of small, different projects. Generally speaking, neural networks cannot change their network structure even though the modularity structures of target projects are different respectively. Besides, an individual project is too small to train neural network despite the total amount of our dataset is huge. We solve this issue by training our model as prototypical networks [17]. Prototypical networks is a kind of implementation of few-shot learning. We describe them in detail in section 3.3.

3.1.1 Detection Process

Our system detects inappropriate method placements by solving the problem of classifying methods into plausible packages. We show the overview of our detection procedure as Figure 3.1. Our system is divided into two major components: the encoder and the classifier. The encoder is a neural network that converts input methods into vector representations. The classifier is not a neural network and is created for each project in order to capture the difference in design conventions. We describe how our system works during the inference phase, hence we suppose the encoder is already trained and its parameters are fixed. The detail of the training process is shown in section 3.3.

We use two sets of data during the inference process, reference set and target set. We assume that the reference set is existing source code in the repository and the target set is newly committed code differences, etc. In this figure, for the sake of simplicity, the reference set consists of three methods for each package and the target set consists of only one method. The reference set is used for capturing the correspondence between concerns

and package, and the target set is the detection target, which means our system decides whether the current placement of the target is correct or it should be moved for another package.

We describe the inference procedure below. We first convert all the methods in both reference and target set into vectors using the encoder. Then we compute the package-level feature for each package out of the methods each package encompasses. The classifier receives the package features and the vector representations of the target set, and trains the parameters of it using the reference set. Finally, the classifier outputs the probabilities of which package the target method should belong to. The package with the highest probability is regarded as the classification result. If there is a difference between the current placement of the target method and the classification result and the probability exceeds a certain threshold, the target is detected that its placement is inappropriate and should be moved to the package that the classification result indicates.

3.1.2 Use Cases

Our system has two main use cases:

1. partial detection for newly committed code difference
2. checking the whole project

Our system requires the reference set and the target set for detection. During the partial detection, as we mentioned before, we regard the whole existing code base as the reference set and the methods included in the committed differences as the target set. We can implement this diff-check system as a CI tool or an IDE plugin to contribute to developers' productivity.

When it comes to checking the whole project, we have to divide all methods in the project into the reference set and the target set, like 80% for reference and 20% for target. The reference set is used for training the classifier. We can infer whether placements of the target set are correct or not. In this way, however, we can check only 20% of the target project. Thereby we have to rotate the role of reference and target. Specifically, we perform detection by swapping the reference set and the target set so that all data are included in the target set at least once. We can check the whole project in this way.

3.2 Model Details

We have two major components in our detection system: the encoder that converts textual information of methods into vectors and the classifier that classifies the output of the encoder into plausible packages (Figure 3.1). We describe how they work and how to implement them in detail in this section.

3.2.1 Encoder

The encoder is the neural network model. In this study, we use a model that modifies the Code2vec encoder [16] to make use of the type information of source code.

Code2vec is a model that embeds AST (Abstract Syntax Tree) obtained from a method into vector representation (Figure 3.2). An AST has textual and structural information of a method. We omit the information of method name before embedding as original code2vec does. This is originally because a code2vec model is trained by the task of method name prediction, and we do the same thing because we assume that even if the method name changes, the semantic meaning of the process performed by the body of the

```
String[] reverseArray(final String[] array) {
    final String[] newArray = new String[array.length];
    for (int index = 0; index < array.length; index++) {
        newArray[array.length - index - 1] = array[index];
    }
    return newArray;
}
```

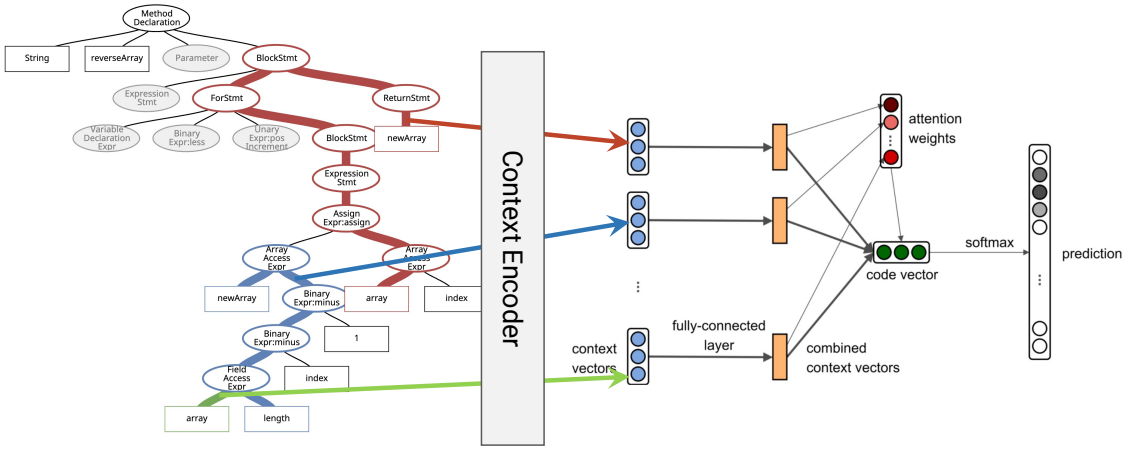


Fig. 3.2. The overview of code2vec encoder

method should not change.

The input format of the code2vec model is not the tree structure of AST but randomly sampled n triplets of two leaf nodes and their connecting path on the AST, which is called the *AST context*. On the AST of Figure 3.2, three colored AST contexts are selected. We selected $n = 200$ in our evaluation in chapter 4 as original code2vec authors did. Individual AST context is represented as tuple of $(w_s, w_t, r_1, \ldots, r_l)$. w_s, w_t are the terminal nodes of both ends of the path and correspond to one of the tokens of the original method. $r_1, \ldots, r_l$ are the l nodes that connect two leaves, and they correspond to structural nodes of AST that represent expressions or statements. For example, on a blue path in the figure 3.2, w_s corresponds for `newArray` and w_t corresponds for `length` respectively, and a path $r_1, \ldots, r_l$ is represented as a sequence of 4 elements: `ArrayAccessExpr`, `BinaryExpr:minus`, `BinaryExpr:minus`, and `FieldAccessExpr`. The role of the encoder is computing n embedded vectors $\mathbf{z}_1, \ldots, \mathbf{z}_n$ out of AST contexts and use them for obtaining distributed representation of a method $\mathbf{v} \in \mathbb{R}^d$.

We embed a path and leaf nodes of an AST context respectively using the context encoder and regard their concatenation as *context vector* (Figure 3.3). We utilize Bi-directional LSTM in encoding paths that is adopted in the code2seq model [15], the succeeding study of code2vec. This mechanism is introduced by the code2seq study as an improvement of code2vec. We can obtain a path encoding function $encode\_path(r_1, \ldots, r_l)$ in the following way:

$$h_1, \ldots, h_l = LSTM(E^{nodes}_{r_1}, \ldots, E^{nodes}_{r_l})$$

$$encode\_path(r_1, \ldots, r_l) = [\overrightarrow{h_l}; \overleftarrow{h_1}]$$

where E^{nodes} denotes the embedding matrix for the words that appear in AST nodes.

The encoding procedure of leaf nodes also follows the code2seq way, but we extended the model to include the type information of source code as input. We first split the

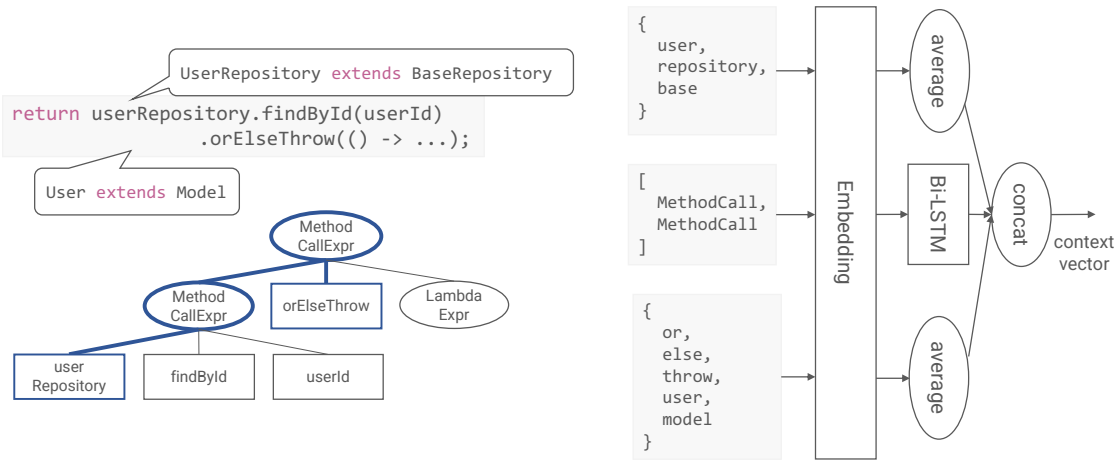


Fig. 3.3. The structure of the context encoder

input word w into subtokens using $split(\cdot)$ function based on the upper-case part of the camel case. For example, the method name `getCurrentTime` can be splitted into three subtokens: `get`, `current`, and `time`. Then we compute $types(\cdot)$ function to obtain a set of types correspond to w. If w is a token that represents for any of class name, interface name, variable name, or parameter name, $types(\cdot)$ returns following set:

- the type name w has
- the superclass name w directly extends
- the interface names w directly implements

Otherwise, the result will be an empty set ($types(w) = \emptyset$). Likewise, all type names are subtokenized at this time as well as word information. For example, in case w is an instance of `ConcreteModel` and `ConcreteModel` extends `AbstractModel` and implements `Serializable`, the type name set will be $types(w) = \{concrete, abstract, model, serializable\}$. We can compute an encoding function of leaf nodes $encode\_token(w)$ using this $types(w)$ function and embedding matrix $E^{subtokens}$ for the words that appears in subtokens:

$$S = split(w) \cup types(w)$$

$$encode\_token(w) = \sum_{s \in S} E_s^{subtokens}$$

Then we can compute a distributed representation $\mathbf{z}$ of a single AST context by following way:

$$\begin{aligned}\mathbf{z} = tanh(W_{in}[&encode\_token(w_s);\\ &encode\_path(r_1, \ldots, r_l);\\ &encode\_token(w_t)])\end{aligned}$$

where W_{in} is the learnable parameters of the neural network.

We have to aggregate these n distributed representations of AST conetxts to obtain a single vector of the whole method. The aggregation method is similar to the original code2vec model. We compute the whole method representation $\mathbf{v}$ as a weighted average of context vectors. The weight is computed by soft attention mechanism. We introduce attention vector $\mathbf{a} \in \mathbb{R}^d$, which is a learnable parameter of the neural network and is

randomly initialized before training. The attention weight α_i for i-th context vector $\mathbf{z}_i$ is computed as follows:

$$\alpha_i = \frac{\exp(\mathbf{z}_i^T \cdot \mathbf{a})}{\sum_{j=1}^{n} \exp(\mathbf{z}_j^T \cdot \mathbf{a})}.$$

Consequently, we can obtain the distributed representation of the target method by computing the weighted average of context vectors:

$$\mathbf{v} = \sum_{i=1}^{n} \alpha_i \cdot \mathbf{z}_i.$$

3.2.2 Classifier

The classifier internally holds the representative points of each package and makes a decision as to which package the input method belongs to. This is the exactly same classifier that ordinal prototypical networks use. This classifier is not a neural network, hence it does not need to be trained with a large dataset in advance. Instead, internal representative points must be computed for each project and for each package that the project has.

We can obtain the classifier for project u in the following way. The training data of the classifier is the reference set of target project u, and is given as $D = \{(x_1, y_1), ..., (x_N, y_N)\}$, where x_i denotes the source code of method m_i $(i = 1, 2, \ldots, N)$ and $y_i \in \{1, \ldots, K\}$ denotes a label of a package that encompasses method m_i. All samples belonging to the package k are denoted as D_k. Then we can compute the representative point D_k of package k:

$$\mathbf{c}_k = \frac{1}{|D_k|} \sum_{(x_j, y_j) \in D_k} f_\phi(x_j) \tag{3.1}$$

$f_\phi(\cdot)$ is the function that represents for aforementioned encoder, which means $\mathbf{v}_i = f_\phi(x_i)$. ϕ denotes the learnable parameters of the neural network. This computation is based on our assumption that the package's concern is represented as the semantic information of methods the package encompasses. This is why we compute representative points from vectors of their including methods.

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

$$p_\phi(y = k \mid q) = \frac{\exp\left(-dist(f_\phi(q), \mathbf{c}_k)\right)}{\sum_{k' \in K} \exp\left(-dist(f_\phi(q), \mathbf{c}_{k'})\right)} \tag{3.2}$$

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

3.2.3 Detection

After the classification, we have to determine whether the input method should be moved or not. Given p_ϕ for each package, our system regards the package that has the highest probability is the plausible package for the input method. Let $\hat{k} \in K$ be the predicted package and $k \in K$ denotes the current package of the input method. If both $k \neq \hat{k}$ and $p_\phi(y = \hat{k}) > \theta$ are fulfilled, our system detects the input method should be moved from k to $\hat{k}$, where $\theta \in [0, 1]$ is the detection threshold. We determine the value of θ in an empirical way. In our case study conducted in section 4.3, we set θ where the performance of evaluation in section 4.2 is maximized.

3.3 Training Method

3.3.1 Few-shot Classification

We utilize few-shot learning techniques in our task, especially called few-shot classification. We introduce the fundamentals of few-shot classification in this section.

Few-shot classification is a task that adapts a classifier to new classes that are not seen in a training dataset, with a limited amount of data samples of the new classes. This is an extremely challenging task because deep neural networks generally require a huge amount of samples for their training. Generally, if neural networks are trained with few training samples, they must indicate symptoms of overfitting and cannot be generalized for tasks that do not exist in the training samples. On the other hand, humans can easily adapt to new classes with very few examples. One possible explanation is that humans have the skill of transferring their prior knowledge to unseen tasks. Based on this assumption, several studies succeeded in adaptation with an approach that uses a large dataset in their pre-training phases and a small dataset in their training phases [39, 17, 40].

During the pre-training phase, a neural network is trained by repeating classification tasks with few data. The provided data has different classes for every iteration, whereas commonly used neural networks are trained by a dataset that has uniform classes. Hence, the pre-training dataset should be a large set of small, different datasets. Note that a large amount of data is required in few-shot classification as well as standard neural networks even though few data is required for the training and the inference phase. The parameters of the neural network are optimized to be able to classify unseen classes with few data during this process.

The separation of pre-training and training is introduced by Vinyals *et al.* as matching networks [39]. They worked on one-shot image classification and achieved promising results. One-shot image classification is the task of classifying an input image to its correct class based on the information of only one reference image given for each class. After this work, a number of variants are proposed, and some of them made significant progress in this kind of task such as prototypical networks by Snell *et al.* [17] and relation network by Sung *et al.* [40].

Few-shot classification was originally developed in the image processing field. It has been widely used for few-shot (or sometimes zero-shot) image classification. Recently, several studies applied this technique for the language processing tasks [41, 42]. However, applications for tasks relevant to source code have not yet been made. There is an attempt of applying few-shot learning techniques for program synthesis [43], but what they did in this study is only preparing a dataset for few-shot learning in program synthesis. Therefore, the model has not yet been developed using this dataset.

3.3.2 Prototypical Networks in Our Task

We use prototypical networks as our few-shot classification model. Its pre-training is conducted by meta-learning and using the large dataset collected from GitHub. We discussed the problem of the dataset in section 3.1 that the dataset consists of a small, different set of projects and is not suitable for training neural networks in a uniform setting. However, as we discussed in the previous section, these characteristics are suitable for few-shot classification.

By doing meta-learning in the pre-training phase, we can expect that the classifier specialize well in the target project. As we mentioned before, we have to train the classifier

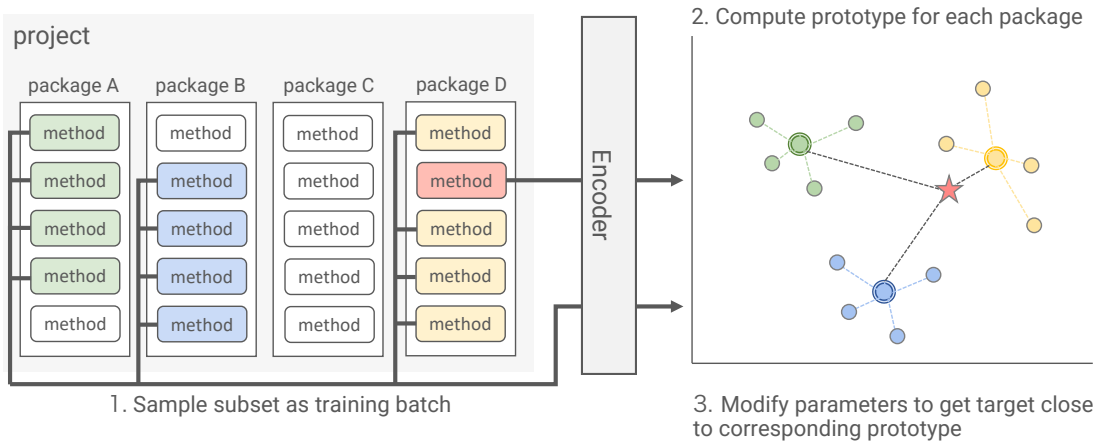


Fig. 3.4. An episode of meta-learning in 4-shot 3-way setting

for each project. Such a classifier is no longer a model that always uses the same rules for every project, but one that flexibly changes the rules for each project. The naming and architectural design conventions are totally different for each project. Our classifier can be expected to consider such project-specific differences.

Among several few-shot classification techniques, we select prototypical networks as our model. This is because the computational simplicity of prototypical networks is suitable for detecting bad code from source code differences. We assume our model is used for checking committed differences as explained in section 3.1. Our model uses existing source code as training data and regards code differences as classification targets. The problem here is that code differences are not only additions but also include modifications and deletions, which means the training data of our model is sometimes modified and deleted. Prototypical networks have an advantage in handling them with low computation cost because its classifier's parameters are just a set of averages of each class (Equation 3.1). For example, when a method in class k whose vector representation is $\mathbf{v}$ is deleted, we can simply update $\mathbf{c}_k$ as follows:

$$\mathbf{c}_k \leftarrow \frac{\mathbf{c}_k \cdot |D_k| - \mathbf{v}}{|D_k| - 1}.$$

Unlike other few-shot classification techniques, we do not have to keep all method vectors to handle modifications and deletions by utilizing prototypical networks.

Meta-learning

Throughout the meta-learning process, we train the learnable parameters ϕ in the encoder f_ϕ. we can obtain a good classifier out of a small dataset of a single project using meta-learning. In meta-learning, we create a small subset of projects from a large number of projects for every training iteration and repeat the training task of creating a classifier from the small subset and classifying it. In brief, meta-learning is the process to learn encoder parameters ϕ to create a good classifier out of a small amount of data.

We illustrate an *episode*, a single cycle of mini-batch training in the context of meta-learning, as Figure 3.4. We choose a target project out of the training dataset for every episode. A target project is chosen probabilistically according to their number of samples. Given a project set U, the probability in which the project $u \in U$ is chosen as a target

project is

$$p(u) = \frac{|D_u|}{\sum_{u' \in U} |D_{u'}|}$$

, where D_u denotes all data samples included in the project u. In this example of Figure 3.4, the target project has 4 packages and each of them has 5 methods respectively. Then we randomly choose K packages from the target project and randomly choose N methods for each K package. In this example, $K = 3$ and $N = 4$. This subset of the target project is called *support set* in the context of meta-learning. Support set is used as the reference set of a classifier to compute representative points of each package according to the equation 3.1. We also select query method q from K packages respectively, taking care not to duplicate the support set. After calculating probabilities based on the equation 3.2, the gradient descent proceeds to minimize the following loss function:

$$J(\phi) = -\log p_\phi(y = k \mid q).$$

We can obtain the encoder by iterating such episodes. This learning setting is called N-shot K-way meta-learning.

3.4 Discussion

Our proposed model is an application of an existing model for the detection task of inappropriate method placements. The structure of our model is a combination of some existing methods. The encoder model is based on a series of studies by Alon *et al.* [16, 15] and the entire structure is based on prototypical networks [17]. However, there are no other studies that focus on modularity-related tasks as downstream tasks of these neural-network-based models.

There are some other studies in this field that adopt a learning-based method using neural networks. Pigazzini tried to detect architectural smell using neural network [12], but its precise evaluation has not yet been done. Liu *et al.* proposed a method of feature envy detection using neural network [11]. Although these methods only focus on learning global rules and do not focus on individual optimization like our classifiers for each project, the work by Liu *et al.* is the most closer to ours in point of the task setting and the model structure. Thus, we conduct an experiment of comparing the performance of our model to their model in section 4.2.

Our model utilizes textual information of source code to detect inappropriate method placements in a concern-based way. There are many studies that utilize textual information proposed as we described in chapter 2. As textual-based metrics, LSI and C3 have been used for code smell detections and concern-based refactorings [32, 34, 6, 35]. Although these methods and our method share the textual feature, the primary difference is that we use automatically extracted distributed representations of source code in our detection task while existing methods use their own manually computed metrics out of textual features. In addition to the textual information, we also embed the AST structure into distributed representations. Recent deep learning models embed structural information of source code such as AST information, data flow, control flow, and so on. It results in great improvement on performance in bug detection [9, 10], code summarization [44, 8], method name suggestion [16, 15, 9], and some other software engineering tasks. However, such models have not yet been introduced in modularity-related tasks.

We aim to detect indicators of architectural design problems in this study by focusing on detecting inappropriate method placements. Existing approaches have also tried to detect architecturally bad code in indirect ways, *i.e.*, by detecting code smell [45, 38].

However, our model can point out problematic methods and where to move it directly. In this respect, our study has an advantage over existing studies.

Chapter 4

Evaluation

4.1 Overview

Our evaluation is three-fold. Each of them corresponds to the following three research questions.

- **RQ1:** Can our model achieve higher performance in the detection task than other methods?
- **RQ2:** Are move method instructions produced by our model really valid and architecturally-relevant ones?
- **RQ3:** Are our model designs really appropriate?

We first evaluate performance in detecting artificially-created wrong method placements and compare the performance of our model to it of an existing method. Second, we conduct a case study of applying our system for real-world applications to confirm our system can really detect architecturally-relevant bad code. At last, we examined our model designs are really appropriate by doing an ablation study. The following subsections explain experimental methodologies commonly used throughout the evaluation.

4.1.1 Dataset

We use *java-large* dataset with some modifications. This dataset is created by Alon *et al.* and used in the evaluation of code2seq [15]. They collected more than 9,500 Java projects from GitHub in order of their star count. We selected this dataset because we can expect these projects have better modularity and naming conventions than low-starred ones.

We have once bundled the three categories of the dataset (training, validation, and test), and divided them again. This is because original projects in the validation and test dataset are not suitable for our evaluation. Originally this dataset is created for the task of method name prediction, not for modularity affairs. Some of the projects in the validation and test dataset have very few packages and have no design principles. These projects are inappropriate for the evaluation of our task. Consequently, we bundled them and picked suitable projects as the validation and test dataset. We listed the projects with their numbers of packages are between 15 and 100, and we randomly selected 250 projects as the validation set and 300 projects as the test set among these projects. Projects of this size are likely to be developed with a consciousness of modularity, and thus are suitable for our task.

Several metrics of our dataset are shown in Table 4.1. Our dataset has basically three categories, training, validation, and test. The category user-projects is our own dataset for our case study and consists of projects other than ones in java-large dataset. The detail of user-projects is described in section 4.3.

Table 4.1. Dataset

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

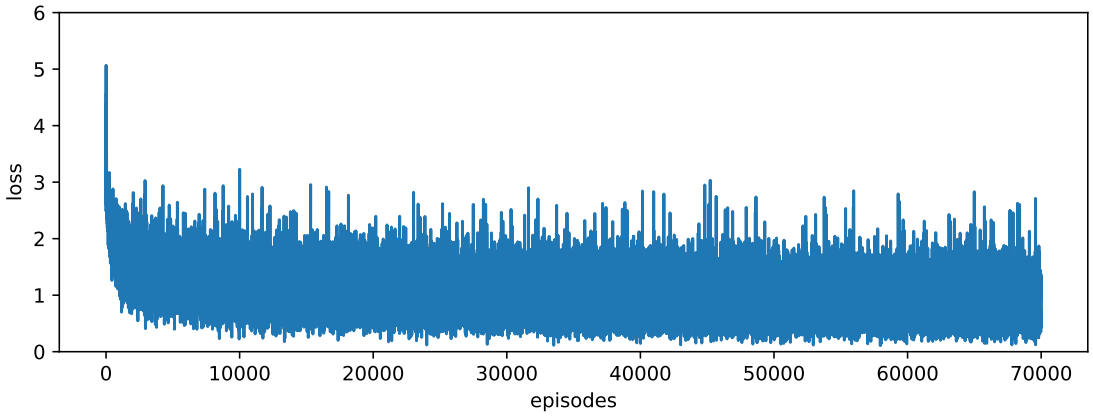


Fig. 4.1. Training loss

4.1.2 Preprocessing

We have some preprocessing tasks for our dataset. We first remove source code for automated testing as possible. This is because they are not relevant to architecture designs. We removed them by filtering classes that have a subtoken `Test` in their name. Besides, we omit so-called boilerplate code since it is useless to consider placement errors for those methods. Boilerplate code often appears in Java systems regardless of architectural affairs. Creating boilerplate code in Java systems is just a convention and has no design-related meanings. We refer to getters, setters, overridden methods of `Object#equals`, `Object#hashcode`, and `Object#toString` as boilerplate. In addition, all comments that appear in the source code are removed during preprocessing.

We create word dictionaries for the use of word embedding in the neural network. We have two types of dictionaries: the collection of words that appear in the text of methods and appear in AST nodes. Each of them corresponds to $E^{subtokens}$ and E^{nodes} introduced in the section 3.2. The word set from 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 minor words with special word `<UNK>`. This replacement is a typical way used in NLP (Natural Language Processing) tasks, which contribute to preventing models from overfitting.

4.1.3 Model Training

We pre-trained our model with meta-learning for approximately 70,000 episodes. Our model is trained on an NVIDIA A6000 GPU, and it took 27 hours. The training loss is shown in Figure 4.1. Unlike ordinary neural network training, the training loss in our case gradually decreases with oscillations. This is because our model handles different projects

Table 4.2. Hyperparameters of our model

Name	Value
subtoken embedding dimensions	128
path embedding demensions	128
code vector embedding dimensions	328
dropout rate before FC layer	0.25
max path length	8
max path width	2
max number of path-contexts	200

for each episode. As we mentioned before, the projects in our dataset are not uniform. They have different architecture designs and different conventions. Therefore, depending on the project selected, the loss can be far off from the surrounding average.

We set the learning rate of our model as 0.001 and the meta-learning configuration as 10-shot 20-way. For each episode in meta-learning, we select 5 samples for each package as query data. Other hyperparameters of our models are shown in Table 4.2.

10-shot 20-way meta-learning causes a problem of dataset usage limitation. This configuration requires more than 200 samples including queries for every episode. However, there are smaller projects than the required size in the training dataset. In our configuration, we cannot use these projects as training samples. This is the limitation of our learning method. In fact, the number of projects used for our meta-learning is 955. Although this is only 10.6% (= 955 / 9005) of the overall dataset based on the number of projects, this is also 72.7% (= 5,682,782 / 7,821,121) based on the number of methods since the project used for meta-learning are relatively bigger ones.

4.2 Comparison between Feature Envy Detection

We first examined our model in the detection performance. In addition to measuring the performance, we compare our model to another existing model to verify our proposed method is required to solve this task. However, there are no other methods proposed for the exact same task we work on. Therefore, we decide to adapt a model from a previous study that has tackled a similar task to our task and compare their performance.

4.2.1 Reference Model Description

Among previous studies, there is a similar study by Liu *et al.* [11]. The purpose of this model is to detect feature envy using a neural network. A number of studies have proposed methods to detect feature envy [46, 29, 11]. Most of these studies utilized structure-based metrics that can be computed from attribute references. Among them, the study by Liu *et al.* is unique in point that they utilize not only attribute references but also semantic features such as method and class names as alternatives of textual features of method bodies. Moreover, the most unique point of their study is they utilize a neural network in the feature envy detection task.

We deem the feature envy detection model by Liu *et al.* can be a good baseline for our evaluation. The feature envy detection task is similar to our task in the sense that it detects inter-class move method opportunities. Besides, the techniques used in the model is close to ours since it also utilizes neural network and semantic features.

The feature envy detection model is the binary classification model. The model outputs whether the input method should remain in the current class or be moved to another class.

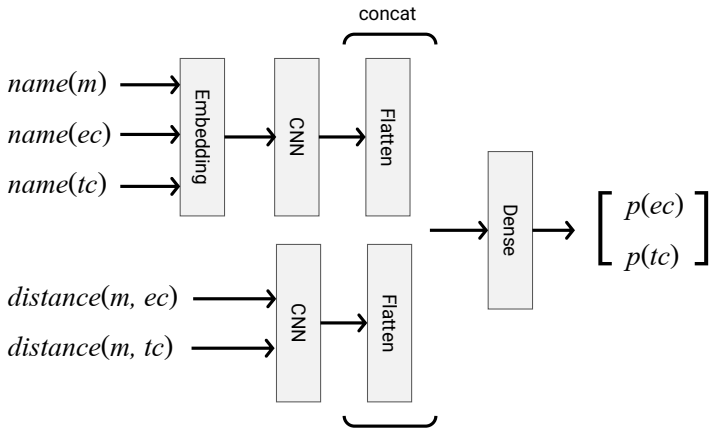


Fig. 4.2. Overview of feature envy detection model

The overview of this model is shown in Figure 4.2. In this figure, m denotes the target method, ec denotes the enclosing class of m, and tc denotes the target class where m might be moved. That is, the inputs are the method name, the class names, and distances between the method and classes. The distance is a structure-based metric proposed by Fokaefs *et al.* [29] that can be calculated out of attribute references. The output of the model is a two dimentional vector, whose first element $p(ec)$ represents for the probability that m belongs to ec and second element $p(tc)$ represents for the one that m belongs to tc. The sum of $p(ec)$ and $p(tc)$ is 1. If $p(tc)$ exceeds 0.5, it is indicated that the target method m should be moved from ec to tc. If there are multiple candidate classes to move, the inference is done for each candidate. After that, we can tell the method should be moved to the class has the maximum score among classes that has more than 0.5.

4.2.2 Task Description

We compare the performance of the models by the task of detecting methods placed in wrong packages. We use the test dataset in the comparison.

We artificially create wrong samples by moving some methods to other packages. The methods to be moved are randomly selected with 1% probability. The destination candidates of method m are the union of the following sets.

- packages encompass methods that call m
- packages encompass methods that are called by m
- packages that have an exact same import statement as the class file where m exists

We assume these packages are more likely to be mistakenly selected than other random packages. The move destinations are selected randomly from this set of each method.

The task in this evaluation is to detect these artificially moved methods. We use precision, recall, F1, and accuracy as evaluation metrics. These metrics can be calculated as follows:

$$precision = \frac{true\ positives}{true\ positives + false\ negatives}$$

$$recall = \frac{true\ positives}{positive\ samples\ in\ the\ dataset}$$

$$F1 = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

$$accuracy = \frac{correct\ recommendations\ for\ true\ positives}{true\ positives}$$

The model outputs 1) whether the input method should be moved and 2) the recommended destination package if the method should be moved. The *correct recommendation for true positives* in the accuracy calculation is the number of accurate recommendations of destination package in the true positives.

The comparison target in this evaluation is the feature envy detection model described in the previous section. However, this model is designed for inter-class misplacements. We adapt this model for inter-package problems by replacing its input with package features and re-train the model with our dataset.

The input of the target model is given by $<name(m),\ name(ep),\ name(tp),\ distance(m, ep),\ ditance(m, tp)>$, where ep denotes enclosing package, which means the package that encompasses the target method m, and tp denotes target package, which means the target package where m will be moved. Given a package p, we can get splitted package name by applying $name(\cdot)$ function. For example, if p represents for the package name `com.foo.bar`, the value produced from $name(p)$ is a string of `"com foo bar"`. $distance(m, p)$ is a metric that describes the relation between a method m and a package p. Let e denote an entity, an attribute or a method encompassed in a package, and S_m denote a set of entities that m accesses or m is accessed from. If m is not encompassed by p, the distance value is computed as follows:

$$distance(m, p) = 1 - \frac{S_m \cap S_p}{S_m \cup S_p}, where S_p = \bigcup_{e_i \in p} \{e_i\}.$$

Otherwise,

$$distance(m, p) = 1 - \frac{S_m \cap S'_p}{S_m \cup S'_p}, where S'_p = S_p \backslash \{m\}.$$

This metric is the one that is simply replaced the original metric's class affairs with package affairs.

We trained this model with our own dataset. Liu *et al.* have made their preprocessing code public, but their software is implemented as an Eclipse IDE plugin and is meant to be operated on GUI. This design is not suitable for processing a huge amount of data. Therefore, we developed our own preprocessing tool that can be operated on CLI and can process in parallel referencing their original soruce code.

In this evaluation, we check each project to detect errors. The way of performing detection is like 5-fold cross-validation as we described in section 3.1.2.

4.2.3 Result

As the evaluation result, our model achieved higher performance than the baseline. We show F1 and accuracy scores with different thresholds (Figure 4.3). The performance of our model is higher than the baseline at every threshold value.

The performance of our model is quite higher in accuracy. This result signifies that, given a method that is placed in an inappropriate package, our model is useful in suggesting the correct package where it should be placed. In terms of model implementation, our model uses textual information of method bodies, whereas Li *et al.*'s model uses only names of methods and packages. Hence we might be able to say that it is necessary to understand the textual features of method bodies in order to determine which packages' concern the target method implements.

On the other hand, about the F1 score, the performance is only slightly higher. This is mainly because of the low precision of our model. We show the PR curve as Figure 4.4.

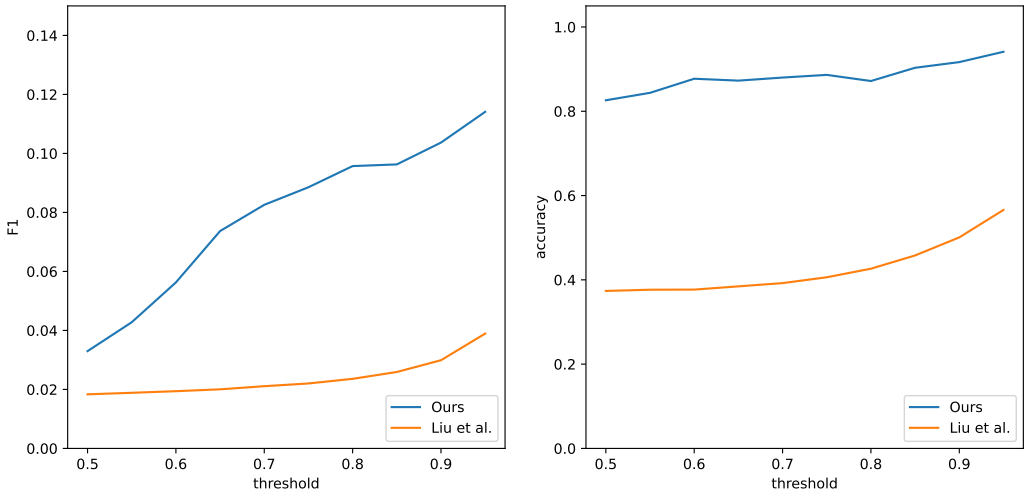


Fig. 4.3. F1 and accuracy with different thresholds

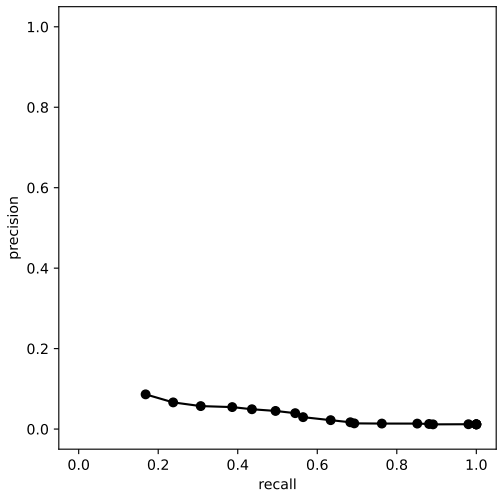


Fig. 4.4. PR curve

We can see the precision values are low everywhere while the recall sometimes achieves high value at certain threshold levels

We consider the cause of this low precision can be attributed to the experimental setup. The reason we suspect is there are many methods that are reasonable to be detected by our model but are not counted as correct detections. Our dataset might contain a small number of problematic code other than ones we artificially created although we deliberately select well-modularized projects. This is inevitable because we have to use noisy datasets unless we manually annotate them. Moreover, some projects have different compilation units in them. For example, library code and its demo application code. Each compilation unit has respective architecture designs. If they are treated as the same project, it will be easy to make false suggestions, such as that library-side methods should be moved to app-side packages. Nonetheless, it is quite proper that our model detects these methods. In this evaluation setting, however, it cannot be counted as a correct detection. Even worse, it would lower the precision.

In conclusion, we confirmed our model is quite useful for determining where to move.

In contrast, because of the noisy dataset, it is not yet clear to what extent our model can detect inappropriate method placements though our model achieved slightly higher performance than the baseline. To clarify it, we manually count the precision on real-world software projects as a case study in the next section.

4.3 Case Study

In order to answer RQ2, we conducted a case study that applied our model for real-world applications. The detection target of this evaluation is all methods included in user-project dataset. The condition of detecting as wrong placement is that 1) the current package of the target method and the inferred package are different and 2) the probability of the inferred package is greater than 0.95, where the highest F1 score is achieved in the previous evaluation.

We aim to detect code that badly affects the architecture design. In this case study, we regard code that matches any of the following conditions as architecturally-problematic code.

- cases that violate general modularity principles
- cases that violate design principles of the adopted framework
- cases that violate project-specific design principles

General modularity principles refer to separation of concerns, information hiding, and low coupling and high cohesion. The framework design principle is, for example, MVC2. Projects that adopt Spring Framework must follow MVC2 design principles such as that controllers should modify data models. In addition to them, projects sometimes have their own design rules because general rules cannot define very detailed rules for each project. Sometimes design principles in these three layers conflict between them. For example, a project-specific design rule that all model modifications are implemented in controller is conflicting with the MVC2 design principle. In such cases, the higher-level principle takes precedence, and this example is judged to be a bad code.

4.3.1 User-project Dataset

We created user-project dataset for this case study. This is the set of projects created by Github *users*. There are two user types in GitHub users, which are organization and user. We collected *users*' projects, not organizations'. We do not use the test dataset in java-large because most of the projects contained in java-large are well-modularized and carefully reviewed ones created by organizations. Such projects are not suitable for the case study since really harmful bad code have already been removed through peer-reviewing. This is the reason why we created the new dataset for the case study.

The user-projects dataset consists of most-starred projects on GitHub created by *users*. The reason we avoid organization projects is the same as java-large dataset. The projects created by users can be expected that are not reviewed so carefully and have more bad code remaining than organization projects. We basically select projects with a high number of stars created within the last 10 years, and excluded projects that overlap with the java-large dataset so that they would not be included in user-projects. We collected more than 7,000 projects in this filtering condition, but it is too much in our case study. Hence we randomly picked 1,000 projects randomly and used them as user-project dataset.

Table 4.3. The result of case study

Category	# of cases
Detected cases	319
Problematic code	51
Correct destination	49
Suspended	40
Excluded	108

4.3.2 Result Summary

The detection result is shown in Table 4.3. Our model detected 319 cases in the user-projects dataset. Among them, we found some cases are easily determined whether problematic or not because of our lack of domain knowledge about the project. We labeled them as "Suspended" and the number of them is 40. We also found that our model detected cases that should not have been detected in the first place. The 108 "Excluded" code refers to such cases. We excluded them from the calculation of detection performance. Finally, we have 171 valid cases (= Detected cases − Suspended − Excluded). Among them, we found 51 cases are really problematic code from the viewpoint of architecture design. Thus, our model's detection precision can be calculated as 0.298 and its destination accuracy is 0.961.

108 excluded cases have some variants. The most common case is ignorance of compilation units. This can often be seen in multi-project repositories. For example, a project that has a library project and its demonstration application project is a typical one. Targetting such repositories, our model often suggests that library methods should be moved to its demo application's package. Although the library code and application code are in the same repository, they have different compilation units. The second common case is a project is not an implementation of specific software, such as chapter-by-chapter works on some material or a catalog of algorithms. These projects are not created with modularity consideration. Due to this, our model often detects these code, but they should be excluded from this case study.

The precision achieved a higher score than the previous evaluation. This is the expected result because we had many noisy samples in the previous one. We have excluded such cases in this evaluation as we described before. Thus the detection performance of our model when applied to real-world projects is expected to be about this level.

4.3.3 Detected Cases

We found 51 architecturally problematic code as a result of the case study. Listing 2.4 and 2.6 are the examples detected by our model. The result shows that our model can detect cases from web applications, Android applications, and many others, regardless of the architecture designs they adopt.

We introduce a few detected cases other than Listing 2.4 and 2.6. The method shown as Listing 4.1[*1] describes the internal detail of the database initialization procedure. Hence, the responsibility of this method should be implemented in `db` package as other database-related code does. However, this method is not placed in `db` package but in the same package as the entrypoint of the application. This is regarded as a concern leakage and

[*1] `https://github.com/deali-axy/minimalpoem`

```
package cn.deali.minimalpoem;  // -> cn.deali.minimalpoem.db

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 4.1. Initialization outside of dao and also an example of feature envy

```
package aspect; // -> client

class TrmConnectionAspect {
    @PostConstruct
    public void afterPropertiesSet() throws Exception {
        NettyClient client = new NettyClient ("localhost", 8888);
        client.start();
        NettyClient.client = client;
    }
}
```

Listing 4.2. Creating singleton instance outside of the class

makes the application inadaptable. If developers need to implement additional entrypoint other than this class, this kind of leakage will be a cause of code clone.

Our detection targets are not completely different from the ones detected using structure-based metrics even though our research motivation is concern-based detection. This method can also be detected by structural metrics. This is because this method is patternized as feature envy since it has more attribute references to other classes than internal references.

The method shown as Listing 4.2 is not only architecturally problematic but also a cause of bugs. Our model suggested this method should be moved to `client` package, where `NettyClient` class is declared. As look out over the project, `NettyClient` is used as a singleton class. The method in Listing 4.2[*2] is the one instantiate its class and set as the static field `client`. This instantiation procedure should be encapsulate in `NettyClient` class. Therefore, the suggestion of our model can be considered as the correct one. However, problems with this method are not limited to design affairs. Since the user of `NettyClient` class sets a singleton instance, it is possible to create a new instance without using that instance in other usage locations.

This is also an example that cannot be solved by simple move refactoring. Our model's suggestion that this method is problematic and should be moved to `client` package is totally correct. However, if the method is moved to `client` package, it is still weak as the implementation of singleton instance. This is because we can instantiate multiple instances

[*2] https://github.com/henkudeluqiqi/gtm

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

package service;
public class FooServiceImpl {
    @Autowired
    private RedisTemplate redisTemplate;
}
```

Listing 4.3. Separation of instance creation and its use in Spring Framework

by calling this method. In conclusion, this method should be moved and re-implemented in accord with *the singleton pattern* [47]. Therefore, move method refactoring cannot completely solve this problem.

4.3.4 Analysis about False Positives

Although our system detected really problematic code from the dataset, there were false positives as well. Analyzing them, we found some patterns difficult to be detected by our system.

The first pattern is the separation of instance creation that is often seen in the project that adopts Spring Framework. We show the example as Listing 4.3. This is an excerpt from a web application project that adopts Spring Framework. Spring Framework provides an easy way to use dependency injection. An instance of `RedisTemplate` is created in the `config` package. `@Bean` is a method annotation provided by Spring Framework, which means the returned value is stored by the framework. Such stored instances are called *beans*. The bean is automatically injected to attributes with `@Autowired` annotation. In this example, `redisTemplate` instance in `service` package will refer to the bean. There is a widely-used convention of Spring Framework that packages all bean creations as `config` package. Thus, the methods in `config` package do not share specific concerns other than bean creation. Our model tends to mistakenly classify these methods into `service`.

The second false positive pattern is that an API class just delegates its function to internal classes (Figure 4.5). This implementation pattern can be seen in library projects and it enables library users to access its functions simply, hiding its detailed implementations. In this case, the API class does not implement its own logic but delegates all implementations to other packages. Because the API class does not have its own concern, our model tends to classify their methods into packages of their delegation destinations.

4.4 Ablation Study

We conducted an ablation study on our model as an answer for RQ3. We adopted the AST-based encoder as described in section 3.2. However, we have other candidates for

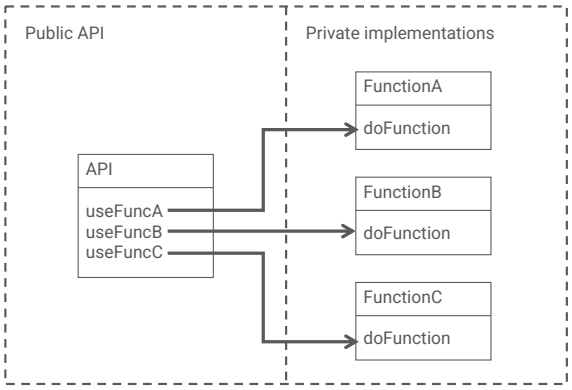


Fig. 4.5. API class provides function accessors and delegates them to internal classes

the encoder such as token-based ones and graph-based ones. In this section, we evaluate them and confirm whether our model design is really appropriate. We also evaluate the effect of subtokenization and utilization of type information introduced in section 3.2.

4.4.1 Evaluation Method

We evaluate models' performance just as classifiers. Our evaluation task is the method-to-package classification. First, we split the test dataset into the 80% reference set and the 20% target set. We use the reference set to train the model and evaluate it by the classification performance of the target set. We input the methods in the target set into the model with their packages masked, and the model outputs the most plausible packages. We compare the results and their original package labels to compute performance metrics. This evaluation process is done in the way of 5-fold cross-validation, which means we perform 5 times evaluation for each project swapping the reference and the target dataset. The average of 5 times evaluation is the performance score of the target project. Hence the average of whole projects' scores is the evaluation score of the target model.

We use top-1 accuracy and set-based hierarchical F1-score for the performance metrics. Each metric is calculated for each project, and the average of all projects included in a dataset is regarded as the score of an encoder. Top-1 accuracy is the proportion of "exact" matches (the actual package that a method belongs to is predicted with the highest probability) against anything else.

Instead of the commonly used F1-score, we use set-based hierarchical F1-score, which we refer simply to as *hierarchical F1* for brevity. It measures the performance of a predictor whose output is structured as trees, such as a Java package that are modularized as a tree structure. Hierarchical F1 compares how close a predicted package is to the actual package based on the similarity between the package trees. For example, if the actual package that a method belongs to is `A1.B.C` and the predicted package is `A2.B.C`, hierarchical F1 tells that the prediction is much closer to the actual package than to `D.E.F`.

The hierarchical F1 we use is slightly modified from the original version [48]. The original version targets multi-label problems (a prediction consists of multiple labels), although the output is a single label (package name) in our task. We calculate our version of hierarchical F1 as follows. Let $T = \{p_1, p_2, \ldots, p_N\}$ be a Java package tree, where N is the number of packages and p_i $(i = 1, \ldots, N)$ denotes each package. For a method x, the actual package it belongs to is denoted as $y \in T$, and a predicted package is denoted as $\hat{y} \in T$. We consider a set Y, a package and its all ancestors in the package tree, using

Table 4.4. Hyperparameters of the Encoders

Model	Name	Value
Code2vec	word embedding dimensions	128
	code vector embedding dimensions	128
	dropout rate before FC layer	0.25
	max path length	8
	max path width	2
	max number of path-contexts	200
GGNN	word embedding dimensions	100
	type embedding dimensions	100
	node embedding dimensions	200
	code vector embedding dimensions	200
	ggnn layers	1
	propagation steps	3
Bi-LSTM	word embedding dimensions	128
	layer dimenstions	256
	code vector dimenstions	256
	number of stacked layers	3
	dropout rate between layers	0.1

an ancestor function $An(\cdot)$:

$$Y = y \cup An(y),\ \hat{Y} = \hat{y} \cup (An(\hat{y}))$$

Given these two sets, we can calculate hierarchical precision (P_H) and hierarchical recall (R_H).

$$P_H = \frac{|\hat{Y} \cap Y|}{|\hat{Y}|},\ R_H = \frac{|\hat{Y} \cap Y|}{|Y|}$$

Hierarchical F1 (F_H) is then given by:

$$F_H = \frac{2 \cdot P_H \cdot R_H}{P_H + R_H}$$

We evaluate our model selection is really appropriate. The model selection here means 1) encoder variants and 2) additional mechanisms we implemented, type utilization and subtokenization. We conduct experiments for three types of encoders, AST-based code2vec, graph-based GGNN, and token-based Bi-LSTM. The hyperparameters for each encoder are shown in Table 4.4. We also evaluated type utilization and subtokenization really improve the performance.

4.4.2 Encoder Variants

Code2vec

Our model design is a kind of an extended model of code2vec. We have implemented some additional features to the original code2vec model such as subtokenization, type information, and code2seq-like encoder designs. Therefore, by comparing our model to the original code2vec model, we can verify our extension really works in this task.

GGNN

As a representative of graph-based encoders, we implemented a GGNN (Gated Graph Neural Network) encoder [9, 49]. GGNN is proposed by Li *et al.* [49], and Allamanis *et*

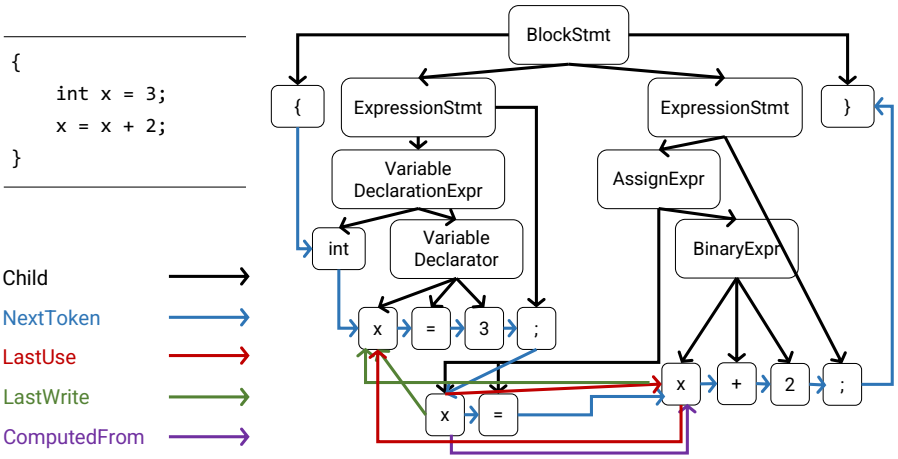


Fig. 4.6. Graph representation of source code

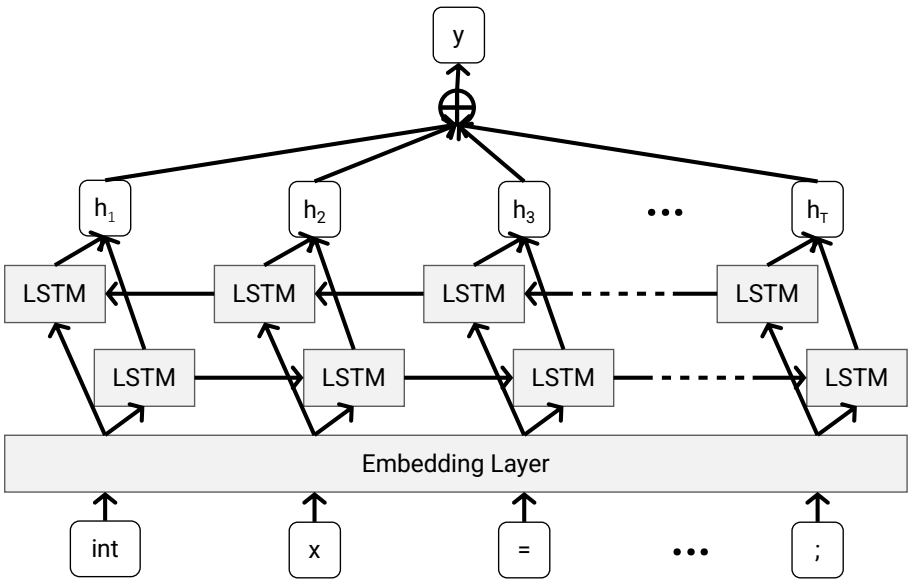


Fig. 4.7. The model structure of Bi-directional LSTM

al. proposed an application of GGGN for source code embedding [9]. Our GGNN model implementation is in accord with the proposal by Allamanis *et al.*.

Allamanis *et al.* proposed the way of representing source code as a graph. We show an example graph representation as Figure 4.6. Their graph representation is based on AST. They added extra edges on the AST such as data flow edges and control flow edges. The graph representation of source code is used as input of GGNN.

GGNN receives a graph and outputs its embedded representation. The encoding method proposed by Allamanis *et al.* can only obtain node embeddings, not an embedded vector of an entire graph. In our task, however, what we need is the embedded vectors of methods. Due to this, we added readout mechanism [49] as the last layer of the GGNN network to obtain entire graph embeddings.

Table 4.5. The result of the ablation study

Variants	top-1 accuracy	F1-score
Ours	**0.652**	**0.779**
w/o types	0.629	0.759
w/o subtokenization	0.605	0.746
Code2vec	0.594	0.762
GGNN	0.585	0.755
Bi-LSTM	0.314	0.559

Bi-directional LSTM

As a representative of token-based encoders, we implemented an encoder that consists of bi-directional LSTM [44] and soft attention mechanism [50]. In short, we call it Bi-LSTM.

We show the model structure of Bi-LSTM as Figure 4.7. The Bi-LSTM encoder's input is a sequence of source code tokens and output is the distributed representation of code. Each token is converted into a vector representation by the embedding layer, and these vectors are the input of bi-directional LSTM. The output of each time step is the input of the soft attention. Applying the attention, they are combined to a single vector, which is the output of Bi-LSTM.

This encoder consists of three layers of stacked bi-directional LSTMs and dropout layers between them and between a bi-directional LSTM layer and its successor.

4.4.3 Result

The experimental result is shown in Table 4.5. Our model achieved higher performance than other model variants. Our model is based on code2vec with some improvements as we described in section 3.3, and its result is much higher than code2vec, which means our improvements contribute to the performance. We can tell both of our improvements, including type information and subtokenization, improve the performance respectively. Moreover, the result shows that the AST-based encoder is more suitable for this task. We can see that even plain code2vec achieved a higher score than other encoder types.

The result has shown that including type information contributes to classification performance. However, in fact, the type information used in our model was incomplete. Specifically, we cannot extract type information for all variables and method call expressions during the preprocessing. This is because we cannot *get all* libraries the target project depends on. Recent software projects usually use Maven or Gradle for dependency management and the way of downloading libraries and building projects varies for each project and is sometimes very complicated. For this reason, we abandoned building the projects in our dataset, and decided to only make use of the type information that can be extracted from the project source code. This decision may have resulted in lower performance of type usage than expected. Nevertheless, the experimental result has shown that it improves performance. Thus, if a dataset that has complete type information is given, the performance of our model will get higher.

4.5 Threats to Validity

Noisy Samples in the Dataset

A possible criticism to our method is that our dataset has several noisy samples. Ideally, it is desired that the correspondences of methods and packages in our dataset are all correct.

Actually, however, our dataset contains a few number of error methods that are placed in inappropriate packages since it is not a manually-annotated dataset but just a collection of projects on GitHub. As we described in section 4.1, we have tried to mitigate this problem by collecting well-modularized projects from GitHub. Generally, neural networks are not so weak to a small amount of mislabeled data. We can also acknowledge the same issues exist in some previous studies [11, 16], but the bad effects of noisy data do not exist or are very limited. Similarly, while better performance could have been achieved with a fully annotated dataset, we did not observe any significant performance degradation due to the noise in the dataset at least.

Representation of Package Feature

We compute package features as a mean of method vectors (Equation 3.1). This computation is based on the assumption that one package has only one concern. However, we can sometimes see a package that has multiple concerns in it. If a package has very distant multiple concerns, the centroid computed by our method might be far from the vector representation of each concern.

We may be able to mitigate this problem by changing the classification criteria, not the distance between a centroid and a method vector but k-NN like ones. However, it is uncertain how the performance will be when such a method is adopted, and the computation cost must get bigger. However, in the current implementation, we have to say that the performance degradation in the presence of multiple concerns is a limitation of our model.

Subjectivity in the Case Study

We had to conduct the case study by only one person due to resource limitations. Therefore, the results of the case study might be subjective and vary from one observer to another. For this reason, we set as clear criteria as possible in the case study. In addition, to make our detection results publicly assessable, we pick up 10 true positive cases randomly and list them as appendix A.

Chapter 5

Conclusion

5.1 Summary

This paper presented a tool for inappropriate method placements detection. Methods placed in wrong packages can be regarded as key indicators of architectural design problems because they increase couplings between two modules. This problem can be solved by move refactoring. Hence, our proposed model points out not only misplaced methods but also plausible packages where they should be encompassed. Our proposed model can be incorporated in IDE and a variety of CI tools. They would prevent design problems in advance and help maintain clean code. Our model also reduces the cost of manually detecting problems by reviewing code, which can also contribute to developer productivity.

We utilized a neural network model to capture the semantic concerns of source code. The model is made of a combination of improved code2vec and prototypical networks. The dataset is collected from GitHub. They are well-modularized but do not have completely correct correspondence between methods and their packages. Due to this, we trained our model in a weakly-supervised way using meta-learning.

We evaluated the performance of our model in three experiments. We first compared the performance to the model proposed by Liu *et al.* and showed that our model outperformed it. Secondly, we conducted a case study to show our model can detect architecturally problematic code out of real-world software projects. Our model detected many cases that badly affects architecuture designs. Finally, we examined our model structure is really plausible by carrying on an ablation study.

5.2 Future Works

Our model has a possibility to fail to handle multiple concerns in a single package as we mentioned in section 4.5. We have already discussed this problem and conclude that this is the limitation of our model. We may be able to mitigate this problem by modifying the model structure. However, this point is outside the scope of this study.

We may be able to mitigate this problem by changing the classification criteria, not the distance between a centroid and a method vector but k-NN like ones. However, it is uncertain how the performance will be when such a method is adopted, and the computation cost must get bigger. However, in the current implementation, we have to say that the performance degradation in the presence of multiple concerns is a limitation of our model.

Our proposed model also has a large room for improvement in performance. We evaluated our model in chapter 4 and showed that our model can really detect architecturally problematic code from real-world applications. However, the result also shows that our model cannot detect all bad code from the input. The improvement of detection perfor-

mance is one of the future research directions.

Another research direction is an application for other programming languages. We only support Java in our proposed method. We believe our model can be applied for programming languages that have method-like functions and package-like module systems. C# will be a primary candidate for the application since C# has methods and namespaces. However, it is a challenging task to apply our method for JavaScript. This is because JavaScript does not have package-like module systems. Besides, coding conventions of JavaScript are highly diverse depending on what framework does a project adopts. This diversity also leads to difficulty in collecting datasets.

Implementing our model as an IDE plugin or a CI tool is another significant task. It may not contribute to research communities, but software engineering proposals are only valuable when they are implemented and used by developers.

Publications and Research Activities

(1) 依田 和樹, 中丸 智貴, 穐山 空道, 山崎 徹郎, 千葉 滋. 2021. Java システムにおけるパッケージ誤りのニューラルネットワークを用いた検出手法. 日本ソフトウェア科学会第 38 回大会.

References

[1] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 255–258, 2009.

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

[3] Edsger Wybe Dijkstra. *A Discipline of Programming.* Prentice Hall PTR, USA, 1st edition, 1997.

[4] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[5] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Syst. J.*, 13(2):115–139, June 1974.

[6] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. A textual-based technique for smell detection. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016.

[7] Isela Macia, Joshua Garcia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic, and Arndt von Staa. Are automatically-detected code anomalies relevant to architectural modularity? an exploratory analysis of evolving systems. AOSD '12, page 167–178, New York, NY, USA, 2012. Association for Computing Machinery.

[8] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 200–210, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[10] Ying Wang, Hai Yu, Zhiliang Zhu, Wei Zhang, and Yuli Zhao. Automatic software refactoring via weighted clustering in method-level networks. *IEEE Transactions on Software Engineering*, 44(3):202–236, 2018.

[11] Hui Liu, Zhifeng Xu, and Yanzhen Zou. Deep learning based feature envy detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 385–396, 2018.

[12] Ilaria Pigazzini. Automatic detection of architectural bad smells through semantic representation of code. In *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ECSA '19, page 59–62, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Fabian Beck and Stephan 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*, ESEC/FSE '11, page 354–364, New York, NY, USA, 2011. Association for Computing Machinery.

[14] Hani Abdeen, Stephane Ducasse, and Houari Sahraoui. Modularization metrics: Assessing package organization in legacy large object-oriented software. In *2011 18th Working Conference on Reverse Engineering*, pages 394–398, 2011.

[15] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating se-

quences from structured representations of code. In *International Conference on Learning Representations*, 2019.

[16] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[17] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 4077–4087. Curran Associates, Inc., 2017.

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

[19] Alexander L. Wolf, C Fl, Dewayne Perry, Dewayne E. Perry, and Er L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17:40–52, 1992.

[20] Francesca Arcelli Fontana, Vincenzo Ferme, and Marco Zanoni. Towards assessing software architecture quality by exploiting code smell relations. In *2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics*, pages 1–7, 2015.

[21] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pages 51–60, 2015.

[22] Duc Minh Le, Carlos Carrillo, Rafael Capilla, and Nenad Medvidovic. Relating architectural decay and sustainability of software systems. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 178–181, 2016.

[23] Umberto Azadi, Francesca Arcelli Fontana, and Davide Taibi. Architectural smells detected by tools: a catalogue proposal. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 88–97, 2019.

[24] William H. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., USA, 1st edition, 1998.

[25] Júlio Martins, Carla Bezerra, Anderson Uchôa, and Alessandro Garcia. Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, SBES '20, page 52–61, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Maurício Aniche, Gabriele Bavota, Christoph Treude, Marco Aurelio Gerosa, and Arie Deursen. Code smells for model-view-controller architectures. *Empirical Software Engineering*, 23, 08 2018.

[27] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[28] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 403–414, 2015.

[29] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1037–1039, 2011.

[30] Marcelo Serrano Zanetti, Claudio Juan Tessone, Ingo Scholtes, and Frank Schweitzer. Automated software remodularization based on move refactoring: A complex systems approach. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, page 73–84, New York, NY, USA, 2014. Association for Computing Machinery.

[31] Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. Detecting software

modularity violations. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 411–420, 2011.

[32] Adrian Kuhn, StÃ©phane Ducasse, and Tudor GÃ®rba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007. 12th Working Conference on Reverse Engineering.

[33] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 133–142, 2005.

[34] Fabio Palomba, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Andrian Marcus, Denys Poshyvanyk, and Andrea De Lucia. Extract package refactoring in aries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 669–672, 2015.

[35] Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, 84(3):397–414, 2011.

[36] Bruno C. da Silva, Claudio N. Sant'Anna, and Christina von F.G. Chavez. An empirical study on how developers reason about module cohesion. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, page 121–132, New York, NY, USA, 2014. Association for Computing Machinery.

[37] Shinpei Hayashi., Fumiki Minami., and Motoshi Saeki. Inference-based detection of architectural violations in mvc2. In *Proceedings of the 12th International Conference on Software Technologies - Volume 1: ICSOFT,*, pages 394–401. INSTICC, SciTePress, 2017.

[38] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 350–359, 2004.

[39] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, koray kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[40] Flood Sung, Yongxin Yang, Li Zhang, Tao Xiang, Philip H.S. Torr, and Timothy M. Hospedales. Learning to compare: Relation network for few-shot learning. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1199–1208, 2018.

[41] Yujia Bao, Menghua Wu, Shiyu Chang, and Regina Barzilay. Few-shot text classification with distributional signatures. In *International Conference on Learning Representations*, 2020.

[42] Mo Yu, Xiaoxiao Guo, Jinfeng Yi, Shiyu Chang, Saloni Potdar, Yu Cheng, Gerald Tesauro, Haoyu Wang, and Bowen 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)*, pages 1206–1215, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[43] Ferran Alet, Javier Lopez-Contreras, James Koppel, Maxwell Nye, Armando Solar-Lezama, Tomas Lozano-Perez, Leslie Kaelbling, and Joshua Tenenbaum. A large-scale benchmark for few-shot program induction and synthesis. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 175–186. PMLR, 18–24 Jul 2021.

[44] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke 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), pages 2073–2083, Berlin, Germany, August 2016. Association for Computational Linguistics.

[45] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.

[46] Ricardo Terra, Marco Tulio Valente, Sergio Miranda, and Vitor Sales. Jmove: A novel heuristic and tool to detect move method refactoring opportunities. *Journal of Systems and Software*, 138:19–36, 2018.

[47] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[48] Aris Kosmopoulos, Ioannis Partalas, Eric Gaussier, Georgios Paliouras, and Ion Androutsopoulos. Evaluation measures for hierarchical classification: A unified view and novel approaches. *Data Min. Knowl. Discov.*, 29(3):820–865, May 2015.

[49] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[50] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

Acknowledgements

修士課程の 2 年間を千葉滋研究室の一員として過ごすことができたことを大変光栄に思います．本研究および修士論文の執筆に際し全面的にご指導いただいた千葉滋教授にまずは深く感謝申し上げます．また，研究方針や論文執筆にあたり多大なる助言をいただいた中丸智貴氏，穐山空道氏，山崎徹郎氏の各氏に感謝申し上げます．最後に，社会人としてのキャリアを一時中断した私の選択を支持し，2 年間の生活を支えてくれた妻の薫実に感謝申し上げます．

A

Detected Cases

Code	Comments

#1[*1]
`class RenderUtils ... {`
`...`
`public static void initRenderMeta(RenderMeta renderMeta) {`
`  XWPFTemplate xwpfTemplate = renderMeta.getXwpfTemplate()`
`      ;`
 	
`  Pattern templatePattern;`
`  Pattern gramerPattern;`
`  try {`
`    Field visitor = xwpfTemplate.getClass().`
`        getDeclaredField("visitor");`
`    visitor.setAccessible(true);`
`    TemplateResolver templateResolver = (TemplateResolver)`
`        visitor.get(xwpfTemplate);`
`    Field configField = templateResolver.getClass().`
`        getDeclaredField("config");`
`    Field templatePatternField = templateResolver.getClass`
`        ().getDeclaredField("templatePattern");`
`    Field gramerPatternField = templateResolver.getClass()`
`        .getDeclaredField("gramerPattern");`
`    configField.setAccessible(true);`
 	
`    templatePatternField.setAccessible(true);`
`    templatePattern = (Pattern) templatePatternField.get(`
`        templateResolver);`
`    gramerPatternField.setAccessible(true);`
`    gramerPattern = (Pattern) gramerPatternField.get(`
`        templateResolver);`
`    renderMeta.setTemplatePattern(templatePattern);`
`    renderMeta.setGramerPattern(gramerPattern);`
`  } catch (Exception e) {`
`    e.printStackTrace();`
`  }`
`}`
`...`
`}`
	From: `com.jump.utils.report`
	To: `com.jump.utils.report.render`
	
	This code is a part of a software that generates MS Word documents using Apache poi library. This utility class implements a rendering concern. Hence, this code should be moved to `render` package. Besides, all methods calling this class are only exist in `render` package.

#2[*2]
`class Timeline ... {`
`...`
`public boolean isAdAvailable(int adGroupIndex, int`
`      adIndexInAdGroup) {`
`  AdPlaybackState.AdGroup adGroup = adPlaybackState.`
`      adGroups[adGroupIndex];`
`  return adGroup.count != C.LENGTH_UNSET`
`      && adGroup.states[adIndexInAdGroup] !=`
`          AdPlaybackState.AD_STATE_UNAVAILABLE;`
`}`
`...`
`}`
	From: `com.google.android.exoplayer2`
	To: `com.google.android.exoplayer2.source.ads`
	
	`AdPlaybackState` fails to encapsulate the operation on `AdGroup` and that leads to message chain. This code increases unneccessary couplig between `source.ads` package. Thus, it seems that this method should be declared as an instance method of `AdPlaybackState`.

[*1] https://github.com/lansetiankong999/reportbuild
[*2] https://github.com/teleplusdev/teleplus-android

#3[*3]

```
class InputReader ... {
...
private void readInput(String messagePrefix,
  ClassPathEntry classPathEntry,
  DataEntryReader dataEntryReader) throws IOException {
  try
  {
    // Create a reader that can unwrap jars, wars, ears,
          and zips.
    DataEntryReader reader =
    DataEntryReaderFactory.createDataEntryReader(
          messagePrefix,
    classPathEntry,
    dataEntryReader);

    // Create the data entry pump.
    DirectoryPump directoryPump =
    new DirectoryPump(classPathEntry.getFile());

    // Pump the data entries into the reader.
    directoryPump.pumpDataEntries(reader);
  }
  catch (IOException ex)
  {
    throw (IOException)new IOException("Can't read [" +
          classPathEntry + "] (" + ex.getMessage() + ")")
          .initCause(ex);
  }
}
...
}
```

From: `proguard`
To: `proguard.io`

`DataEntryReader` and `DirectoryPump` are declared in a lower-level package `proguard.io`. This class is located in a higher-level package `proguard`, but implements a procedure using these classes. The coupling between two packages can be decreased if this method is moved to `proguard.io`.

#4[*4]

```
class Commander ... {
...
@Override
public void setCommunicationInterface(OutputStream
      toOBDStream, InputStream fromOBDStream) {
  this.outputStream = toOBDStream;
  this.inputStream = fromOBDStream;
}
...
}
```

From: `io.github.macfja.obd2`
To: `io.github.macfja.obd2.commander`

This class implements `CommanderInterface` despite the interface declaration is located in the child package `obd2.commander`. This implementation can be a cause of 2-modules cyclic dependency.

#5[*5]

```
class OptionsController ... {
...
private String getOldIP() {
  return ipMonitor.getLastIP().equals("") ? "[OLD_IP_HERE]
        " : ipMonitor.getLastIP();
}
...
}
```

From: `controller`
To: `model.ipmonitor`

This code should be move to `IpMonitor` class because all attributes appears in this method are ones in `IpMonitor` class. This method can be encapsulated in `IpMonitor` class.

[*3] https://github.com/w296488320/nullproguard
[*4] https://github.com/macfja/obd2
[*5] https://github.com/pupi1985/ipmonitor

#6[*6]

```
class MainActivity ... {
...
private void startAllServices(){
  Intent intent = new Intent(this, WebSocketService.class)
      ;
  startService(intent);
  startService(new Intent(this, WatchOneService.class));
  startService(new Intent(this, WatchTwoService.class));
  if(Build.VERSION.SDK_INT >=Build.VERSION_CODES.LOLLIPOP)
      {
    startService(new Intent(this, WatchJobService.class));
  }
}
...
}
```

From: `com.fragmentapp`
To: `com.fragmentapp.service`

This method is a private method in `MainActivity`, but is not used currently because all callers in this method are commented out. Now that all `startService` calls are refactored to be scattered in `service` package, this method is no longer needed. Thus, if this method were valid, it would be plausible to move this method to `service` package because this method increases coupling. However, actually, this method should be removed.

#7[*7]

```
class BeanUtil ... {
...
public static Object deepClone(Object objSource) throws
      InstantiationException, IllegalAccessException,
      InvocationTargetException, NoSuchMethodException {
  if (null == objSource) return null;
  Class<?> clazz = objSource.getClass();
  Object objDes = clazz.newInstance();
  Field[] fields = getAllFields(objSource);
  for (Field field : fields) {
    field.setAccessible(true);
    if (field.getModifiers() >= 24) {
      continue;
    }
    try {
      field.set(objDes, field.get(objSource));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  return objDes;
}
...
}
```

From: `top.danny.tools.bean`
To: `top.danny.tools.reflect`

Reflective operation on objects are implemented in `RefrectionUtil` class in `top.danny.tools.reflect` package. It seems `RefrectionUtil` and this method realize the same concern. Hence this method should be moved to `top.danny.tools.reflect` or two packages are integrated.

#8[*8]

```
class XQRCode ... {
...
public static void disableFlashLight() {
  Camera camera = CameraManager.get().getCamera();
  if (camera != null) {
    Camera.Parameters parameter = camera.getParameters();
    if (parameter != null) {
      parameter.setFlashMode(Camera.Parameters.
            FLASH_MODE_OFF);
      camera.setParameters(parameter);
    }
  }
}
...
}
```

From: `com.xuexiang.xqrcode`
To: `com.xuexiang.xqrcode.camera`

`CameraManager` class has a responsibility of handling `Camera` class included in the android standard library. Therefore, `CameraManager` have to encapsulate implementations that use `Camera` instances. However, in this case, `Camera` is leaked to this class.

[*6] https://github.com/1024477951/fragmentapp
[*7] https://github.com/xzgxp/supertools
[*8] https://github.com/xuexiangjys/xqrcode

#9[*9]

```
class CollectionUtil ... {
...
static <R, T> Set<R> map(final Set<T> input, final FN1<R,
      T> mapper) {
  return input.stream().map(mapper::apply).collect(
        Collectors.toSet());
}
...
}
```

From: `org.reactivetoolbox.core`
To: `org.reactivetoolbox.core.lang`

This software project implements its own collection utilities in `core.lang` package. This method is a part of it since `FN1` is the functional interface implemented in `core.lang`. For this reason, this method can be regarded as a leakage of concern.

#10[*10]

```
class UpExgMsgHandler ... {
...
private String parseDateTime(byte[] dateTime){
  int year = ByteArrayUtil.bytes2int(ByteArrayUtil.
        subBytes(dateTime,2,2));
  int month = ByteArrayUtil.bytes2int(ByteArrayUtil.
        subBytes(dateTime,1,1));
  int day = ByteArrayUtil.bytes2int(ByteArrayUtil.subBytes
        (dateTime,0,1));
  int hour = ByteArrayUtil.bytes2int(ByteArrayUtil.
        subBytes(dateTime,4,1));
  int minute = ByteArrayUtil.bytes2int(ByteArrayUtil.
        subBytes(dateTime,5,1));
  int second = ByteArrayUtil.bytes2int(ByteArrayUtil.
        subBytes(dateTime,6,1));
  return "" + year +
    (month < 10 ? "0" + month : month) +
    (day < 10 ? "0" + day : day) +
    (hour < 10 ? "0" + hour : hour) +
    (minute < 10 ? "0" + minute : minute) +
    (second < 10 ? "0" + second : second);
}
...
}
```

From: `cn.com.onlinetool.jt809.handler`
To: `cn.com.onlinetool.jt809.util`

`util` package encompasses byte array parsing and date processing. Based on this, moving this method to `util` package increases its cohesion.

[*9] https://github.com/siy/reactive-toolbox-core
[*10] https://github.com/ch0ice/jt809-tcp-server