An Efficient Design Smell Detection Approach with Inter-class Relation

Hao Zhu, Yichen Li, Jie Li and Xiaofang Zhang*
School of Computer Science and Technology
Soochow University, Suzhou, China
{hzhu721,ycli1024,2027407061}@stu.suda.edu.cn and xfzhang@suda.edu.cn

Abstract*—Code smell indicates the potential designed problems and quality of source code affecting the software maintenance and readability. Hence, detecting code smells in a timely and effective manner can provide guides for developers in refactoring. Existing methods generally treat these code smells from different granularities equally by merely extracting tokens-based or abstract syntax tree(AST)-based code representation, which does not take the diversity of code smells into account, especially when fewer researches concern design smells. To tackle this challenge, we propose Design Smell Detection through Inter-class Relation, which leverages the corresponding design smells features for code smell detection. More specifically, we employ AST-tokens instead of traditional word-tokens or AST to obtain code syntax information from the deep dimension. Meanwhile, we analyze the common structural feature of design smells and propose the inter-class relation among different class files contained in the same package. Moreover, to verify the effectiveness of our proposed method, we carry out extensive experiments with various settings on our new dataset and the results demonstrate that our method outperforms state-of-the-art methods by up to 31% in terms of F1-measure of all code smells. The code is available at: *https://github.com/xzb777/designSmellUML

***Index Terms*—Code smell, code representation, inter-class relation, deep learning**

I. Introduction

Code smells are indicators of poor coding potentially affecting software quality and maintenance [1]. Therefore, code smell analysis, which is developed to assess and improve software quality by detecting and removing code smells, is of great importance in software design. Software engineering researchers have conducted extensive researches on code smells, including their definition, causes, effects, detection and refactoring [2].

Researchers have devised a variety of methods for detecting code smells, including traditional metric-based methods [3], [4] and heuristic-based methods [5]. As an example, Moha et al. [6] devised DECOR to define code smell detection rules with a specific language. Nevertheless, the majority of these methods suffer from developer's subjective illustration and threshold set to identify smelly instances. To address this issue, machine learning methods [7], [8] and deep learning methods [9] have recently attracted increasing attention for their efficiency in extracting hidden patterns from large amounts of data for prediction. Guggulothu et al. [10] apply RandomForest to code smell detection with tradition metrics. Xu et al. [11] adopt the syntax information in AST to optimize the code smell detection from different granularities. Li et.al [12] apply a hybrid model with comprehensive code features to multi-label code smell detection. Though remarkable performance has been achieved, these methods all focus on implementation code smells like *Long Method*, *Missing default*, ignoring design smells to some extent. For larger-granularity design smells such as *Broken Hierarchy* and *Insufficient Modularization*, they still require careful consideration which usually involve the organization and relation between classes in a software system.

In this paper, we seek to develop a simple and scalable technique to detect design smells. Token-based methods [11] and AST-based methods [13] have had outstanding performance in implementation smells detection with tokens or AST. The rendering of design smells is often accompanied by more complex and longer code snippets, which makes the above methods hard to precisely extract features in practice. A direct idea to solve the problem is to optimize extracted features from the deep dimension and make use of the relation between classes in the same package. In addition, existing public datasets on design smells [12] are split into method/class-level code snippets, losing the relation between classes during data preprocessing.

Considering these limitations, we in this paper propose a novel approach **D**esign **S**mell Detection via **I**nter-class **R**elation (DSIR)– that can efficiently extract features in design smells for detection. Specifically, we first parse the source code into AST and obtain a sequence of node tokens by preorder traversal. Then, we create a class-level semantic graph for each target class based on the UML class diagram. To better extract hidden patterns, we separately apply bidirectional long-short term memory network with attention mechanism (BiLSTM) and relational graph convolution network (R-GCN) and combine the outputs of the two models by weight to obtain the prediction. Besides, we split 500 Java projects on GitHub to build a new dataset with class-relation information and conduct our experiments on it. Experimental results illustrate that our DSIR model outperforms the state-of-the-art methods by up to 31% in terms of F1-measure on the three selected design smells. The major contributions of this paper are summarized as follows:

- We present the appropriate feature extraction for design smells and innovatively apply the class-level semantic

DOI reference number: 10.18293/SEKE2023-208

graph based on UML class diagram to obtain inter-class relation for design smell detection.
- We build a new dataset with design smells and preserve class relation for further research at a large granularity.
- We conduct extensive experiments with various settings and baselines. The results demonstrate the effectiveness of the proposed method which improves the F1-measure by up to 31% compared to state-of-the-art methods.

II. Related work

A. Code Smell

Fowler et al. initially introduced the concept of code smell as structures with technical debt. Code smells serve as indications of deteriorating software quality. Based on granularity, scope, and impact, code smells can be classified into implementation [1], design [14], and architecture smells [15]. Previous research has primarily focused on detecting implementation smells, with excellent results achieved in this regard. However, design smells have a more extensive scope, and identifying and refactoring them may require working with a set of classes.

B. UML Diagram

UML (Unified Modeling Language) is a software engineering modeling language used for describing and designing software systems [16], [17]. Class diagrams, as one of the UML diagrams, are particularly useful for describing the attributes, methods, and relation among classes and interfaces in a system. The relationships in a class diagram, such as dependencies, inheritances, aggregations, and associations, can assist software designers in identifying the structural composition and functionality of a system. However, how to apply UML to code smell detection tasks remains a challenge.

C. Motivation

Design smell is usually expressed as more complex and longer code snippets. Existing token-based methods simply treat code snippets as a natural language to get the structural information. Meanwhile, AST-based methods rely on traversing the AST and analyze the nodes and edges to obtain the syntactic information of the code. However, these approaches are challenging as they require a deep and entire understanding of the AST structure or code tokens which is difficult with long code snippets. Intuitively, using AST tokens instead of code tokens can provide a more expressive representation of the source code and it can help to reduce the feature size of the AST-based approach. Furthermore, We consider that introducing the information from class diagrams can better capture inter-class relation which can help the model learn the hidden patterns of complex class snippets. An example can be seen in Figure1. This is a class diagram provides the relation between classes in a program, including inheritance and dependencies. Each box in the diagram represents a class in the program, and the name of the class is at the top of the box. The properties and methods of the class are listed in the second and third sections of the box respectively.

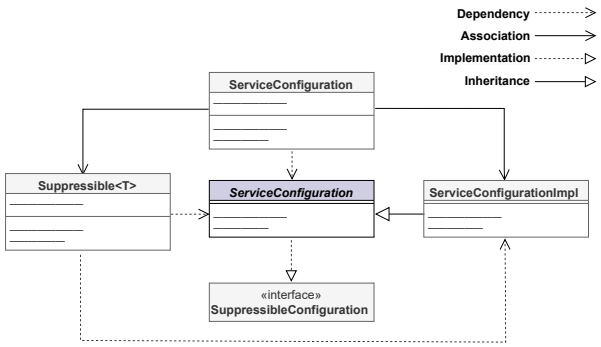


Fig. 1: Example of a class diagram with design smell.

We use the example of an abstract class *ServiceConfiguration* in Figure 1 to illustrate how the class diagram can help us detect design smells. *ServiceConfiguration* is labeled with *Broken Hierarchy* design smell in our dataset. *Broken Hierarchy* refers to the presence of poor inheritance or nesting relation in the code. Then we can see that class ServiceConfigurationImpl inherits from *ServiceConfiguration* because they are connected by a line of inheritance, and class Suppressible and class ServiceConfiguration depend on *ServiceConfiguration*, indicating that *ServiceConfiguration* plays an important role in the class hierarchy and functionality. However, the class Suppressible also depends on ServiceConfigurationImpl, which breaks the functionality and inheritance of parent and child classes which is consistent with the definition of *Broken Hierarchy*. Therefore, to obtain inter-class relation to detect design smells, we create a class-level semantic graph representation based on class diagram information.

III. Methodology

A. Problem Formulation

In this section, we introduce our DSIR method to detect design smells. The overview framework is shown in Figure 2. We are aimed to detect design smells including *Broken Hierarchy*, *Insufficient Modularization* and *Deficient Encapsulation*. To tackle this problem, we divide it into two sub-problems: The first sub-problem involves extracting appropriate features of the source code effectively and we apply the bidirectional long short-term memory network. The second sub-problem involves modeling the relation with relational graph convolutional network between classes based on the class diagrams. Finally, we identify smelly instances based on the fusion model.

B. LSTM Model

We first use Javalang[1] to parse class-level code fragments into their corresponding AST. Then, we obtain all the AST node tokens by preorder traversal and use them as inputs to be fed into an LSTM model. The BiLSTM captures both forward and backward dependencies between tokens, while the attention mechanism focuses on the most relevant parts of the input during the encoding process. Specifically, we use global

[1]https://github.com/c2snet/javalang

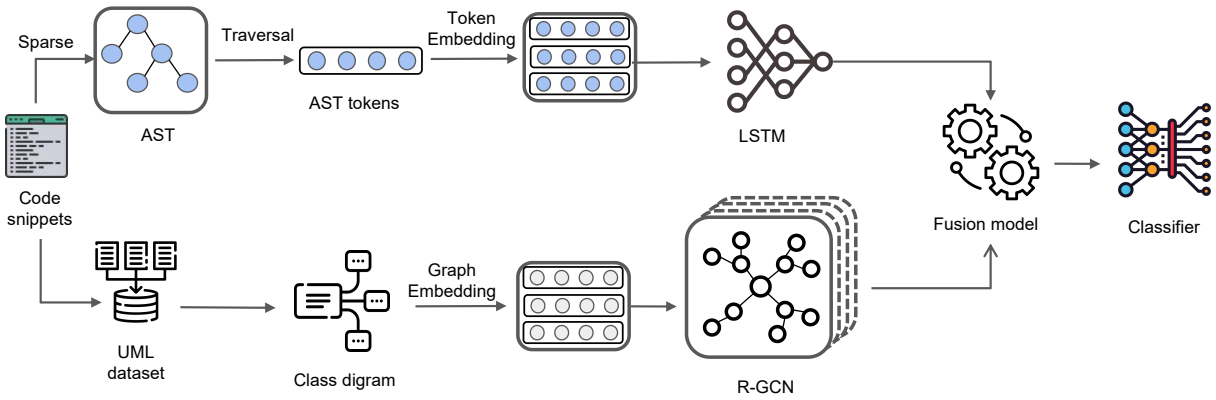


Fig. 2: Overview of our DSIR method.

attention to extract the source context vector c_j by computing the attention weights a_{ij} of hidden state h_i.

$$c_j = \sum_{i=1}^{|x|} a_{ij} h_i \tag{1}$$

where x refers to the AST token sequence. The attention mechanism will assign more weight to the hidden state vectors of important tokens.

$$r_{ij} = h_i * c_j \tag{2}$$

$$y = Sigmoid(W_s r_{ij} + b_s) \tag{3}$$

where r_{ij} represents the relevance score between the i-th hidden state h_i in the source sequence and the context vector c_j for the j-th target token. We then pass r_{ij} through a Sigmoid layer with parameters W_s and b_s to get the output of the model.

C. R-GCN Model

This section presents our approach to detecting design smells using class diagram information to obtain the inter-class relation. We select four common types of relation between classes: dependency, inheritance, association, and implementation, as they are widely used to investigate the metric of inter-class relation in [18]. First, we find the corresponding class diagram for the input class in our dataset, and then we select a set of classes adjacent to it. We then add four common edges to the set of selected classes based on the class diagram. Meanwhile, each class includes its type, name, attributes, and methods in a class diagram. Thus, we extract these information for the selected classes and organize them into an input tuple:

$$\begin{aligned} input &= \langle class\_type, attributes, methods \rangle \\ attributes &= \langle attribute_1, attribute_i, \dots attribute_n \rangle \\ methods &= \langle method_1, method_i, \dots method_n \rangle \end{aligned} \tag{4}$$

Here, $class\_type$ represents the type of the class including concrete classes, interfaces and abstract classes. The class's attributes include different constants or variables, while the class's methods are input in the form of "method_type method_name (method_return_type)", where i represents the $i-th$ attribute or method information of the class and n represents the number of attributes or methods. We input the tuple into a Transformer-based sentence embedding model to obtain the class semantic embedding and use the embedding vectors corresponding to the classes in the set as the node feature matrix. Finally, we use the Python package PyG to convert these information into a graph, which serves as the input to the R-GCN. Relational Graph Convolutional Networks (R-GCN) is designed to process graph-structured data with complex relationships, allowing us to better capture the inter-class relation. The propagation rule for R-GCN is given by:

$$H_i^{(l+1)} = \sigma\Big(\sum_{r \in R} \sum_{j \in N_i^r} \frac{1}{c_{i,r}} W_r^{(l)} H_j^{(l)} \Big), \tag{5}$$

where N_i^r denotes the set of neighbors of node i via edge type r, $c_{i,r}$ is a normalization constant that scales the contribution of each neighbor according to its degree, and $\sigma(.)$ is an activation function. We can stack multiple R-GCN layers by repeating this propagation rule. Our forward model for R-GCN then takes the form:

$$Z_r = \tilde{A}_r \ ReLU(\tilde{A} r X W_r^{(0)}) W_r^{(1)}, \quad \forall r \in R \tag{6}$$

$$Z = \sum_{r \in R} Z_r \tag{7}$$

$$y = Sigmoid(W_s Z + b_s) \tag{8}$$

where Z_r represents the feature vector associated with relation r, $\tilde{A}_r$ represents the adjacency matrix of relation r plus a self-connection adjacency matrix, X represents the input feature matrix, $W_r^{(0)}$ and $W_r^{(1)}$ represent the weight matrices of the input and output layers of relation r, respectively. $ReLU$ is the rectified linear unit function, Z is the vector obtained by summing up all the relation feature vectors, W_s and b_s are the weight and bias of the output layer, respectively, and y is the corresponding binary classification prediction result, which is mapped to the range of $[0, 1]$ by the $Sigmoid$ function.

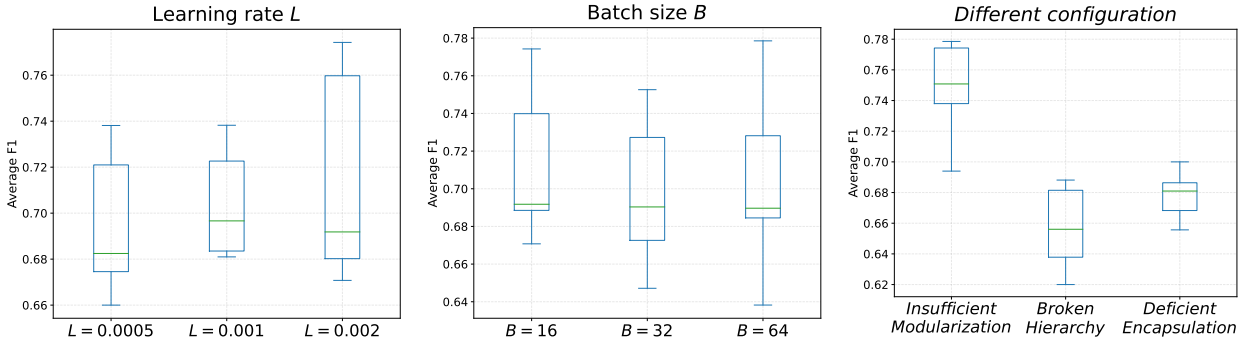
D. Fusion of Model

Assume the outputs of the model are o_1 and o_2 and the hyperparameter k, then the final probability distribution is computed as follows:

$$output = k \otimes o_1 + (1 - k) \otimes o_2 \tag{9}$$

where k is normally equal to 0.5. For both models, we all use binary cross-entropy loss to optimize.

$$Loss(x_i, y_i) = -w_i [x_i log y_i + (1 - x_i) log(1 - y_i)] \tag{10}$$



(a) Box plot of learning rate (b) Box plot of batch size (c) Box plot of model parameters

Fig. 3: Performance of DSIR under different configurations (a) training learning rate L, (b) batch size B, (c) model parameters

where w_i is the parameter for loss, x_i is the i^{th} prediction of the label and y_i is the i^{th} ground truth.

IV. Experiments

A. Dataset Preparation

First, We utilize the same dataset of 500 high-quality Java projects from GitHub as used in [9], covering a variety of functions and diverse application areas. Then we construct a dataset of UML class diagrams for the packages in each project by UMLGraph[2]. Designite [19] is used to detect code smells in our dataset and generate corresponding smell reports. Finally, we select three types of design smells based on their high frequency of occurrence in the 500 high-quality Java projects and label the corresponding class-level code fragments accordingly. We divide projects into three parts, 70% as the training set, 10% as the validation set, and 20% as the test set. Moreover, we carefully balance the dataset to ensure that samples from the same package are not split into different sets and reduce the number of negative samples to the balanced distribution of samples. Table I presents the number of samples used in our DSIR method, as well as the baselines.

TABLE I: Sample distribution of our DSIR dataset

	DSIR dataset					
	Training set		Validating set		Testing set	
Code smells	P	N	P	N	P	N
Broken Hierarchy	5000	5000	369	2518	1628	4145
Insufficient Modularization	1500	1500	137	2750	420	5353
Deficient Encapsulation	5000	5000	881	2006	1899	3874

B. Baselines

In this paper, we select the following three comparative methods as our baseline here:

1) ASTNN Model: The ASTNN model was adopted by [11] to detect multi-granularity and achieved better performance than CNN and RNN in detecting design smells such as *Insufficient Modularization* and *Deficient Encapsulation*.

2) Random Forest Model: This model was used by [10] and performed well in detecting design smells such as *Feature envy*.

[2]https://github.com/dspinellis/UMLGraph

3) LSTM Model: The LSTM model was used in a fusion model by [12] and [20] to extract semantic information from tokens, and achieved great results in detecting implementation smells and *God Class*. In contrast to our approach, we will use the token sequence of code snippets as the input.

C. Evaluation

As the distribution of positive and negative samples in real projects is highly unbalanced, we avoid comparing the accuracy of each model because a model that predicts all samples as negative can still have high accuracy. Therefore, we choose *precision*(*P*), *recall*(*R*) and *F1-measure*(*F1*) as evaluation metrics to account for the imbalanced data. Additionally, we also include *the Area Under the Receiver Operating Characteristic Curve* (*AUC*) to evaluate the performance of our models. The AUC value ranges from 0 to 1, with 1 indicating perfect prediction, 0.5 representing random chance, and values below 0.5 signifying worse-than-random performance. Generally, a higher AUC value corresponds to better classification performance. They are defined as follows:

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{11}$$

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \tag{12}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{13}$$

D. Configurations

Fig 3 presents the performance of the DSIR method under different configurations, including learning rates, batch sizes, and model settings. Based on different performance, we seek the best configuration. Our DSIR model consists of two components: R-GCN and LSTM. For R-GCN, we set the embedding dimensions to be 512 and the number of hidden units to be 300 for *Insufficient Modularization* and *Deficient Encapsulation* and 250 for *Broken Hierarchy*. For LSTM, we set the embedding dimensions to be 200, the hidden dimensions to be 200 for *Broken Hierarchy* and 150 for *Insufficient Modularization* and *Deficient Encapsulation*. For the first sub-model, we apply dropout with a rate of 0.4 to prevent overfitting. For ASTNN, we set two layers and 128 dimensions in the hidden dimension layer and 256

TABLE II: Performance of DSIR and other baselines.

	Broken Hierarchy				*Insufficient Modularization*				*Deficient Encapsulation*				Avg-F1	$\Delta(\uparrow)$
	P	R	F1	AUC	P	R	F1	AUC	P	R	F1	AUC	F1	F1
Random Forest	0.36	0.63	0.46	0.59	0.22	0.23	0.22	0.58	0.38	0.63	0.48	0.58	0.39	0.32↑
LSTM	0.17	0.74	0.27	0.66	0.18	0.34	0.23	0.64	0.36	0.45	0.40	0.70	0.30	0.41↑
ASTNN	0.52	0.48	0.50	0.69	0.64	0.56	0.60	0.79	0.47	0.57	0.52	0.63	0.54	0.17↑
DISR -w/oR-GCN	0.59	0.58	0.58	0.73	0.86	0.65	0.75	0.82	0.75	0.62	0.68	0.72	0.67	0.04↑
DISR -w/oLSTM	0.78	0.60	0.68	0.75	0.47	0.52	0.49	0.71	0.68	0.43	0.53	0.56	0.57	0.14↑
R-GCN -w/oClass-relation	0.49	0.49	0.49	0.65	0.40	0.19	0.26	0.57	0.66	0.38	0.48	0.56	0.41	0.30↑
R-GCN -w/oClass-info	0.42	0.62	0.50	0.65	0.17	0.24	0.20	0.57	0.38	0.44	0.41	0.54	0.37	0.34↑
DISR	0.76	0.63	**0.69**	0.76	0.87	0.70	**0.77**	0.84	0.79	0.60	**0.68**	0.72	**0.71**	/

encode dimensions. Then we choose 80 features and 50 trees in the random forest. Additionally, we use the Adam optimizer algorithm with a 0.002 initial learning rate and the batch size is set to be 16.

V. Performance Overview

A. RQ1:How does our method perform compared to other baselines?

We conduct experiments on our datasets and other baselines. As shown in Table II, DSIR outperforms other baselines, including the state-of-the-art ASTNN, and achieves a 31% improvement in F1-measure, which confirms our initial intuition and demonstrates the effectiveness of our method. Lack of inter-class relation and AST node tokens, other baselines fail to give full play to the same excellent performance as implementation smells in the face of design smells. Specifically, design smells with more complex and longer code snippets will lead to the loss of features for the entire sub-tree if we truncate long traversal sequences of AST. Furthermore, we apply 80 metrics [10] to Random Forest and low AUC values can be observed for all three design smells. We believe that the reason for this is that the complex information of design smells makes it difficult for algorithms to construct rules for detection.

TABLE III: Mappings between Cliff's Delta values and their effective levels.

Cliff's delta	Effective levels
$\|\delta\| < 0.147$	Negligible
$0.147 \leq \|\delta\| < 0.33$	Small
$0.33 \leq \|\delta\| < 0.474$	Medium
$0.474 < \|\delta\|$	Large

TABLE IV: Win/Tie/Loss indicators on F1 values of Random Forest, ASTNN, LSTM and DSIR.

Code smell	DSIR vs Random Forest	DSIR vs ASTNN	DSIR vs LSTM
Broken Hierarchy	<0.05(+Large)	<0.05(+Large)	<0.05(+Large)
Insufficient Modularization	<0.05(+Large)	<0.05(+Large)	<0.05(+Large)
Deficient Encapsulation	<0.05(+Large)	0.05(+Large)	<0.05(+Large)
Win/Tie/Loss	3/0/0	3/0/0	3/0/0

Finally we apply the Win/Tie/Loss indicator as a common approach in prior works for performance comparison [11], [12]. We also conduct Wilcoxon signed-rank test and Cliff's delta test to further analyze the performance of our model and baselines. Table III shows Cliff's delta values($|\delta|$) and the corresponding effective levels. We use a comparison method to determine the Win/Tie/Loss indicator as follows: First, we select a baseline method M. If our model outperforms M with a Wilcoxon signed-rank test p-value < 0.05 and a Cliff's delta ≥ 0.147, we mark our model as a "Win" indicating a statistically significant difference. Conversely, if the baseline method M outperforms our model with a p-value < 0.05 and a Cliff's delta ≥ 0.147, our model is marked as a "Loss". Otherwise, we mark it as a "Tie".

According to the results shown in Table IV, our model performs significantly better than all the compared models in detecting design smells, demonstrating excellent effectiveness.

B. RQ2: What impact does each of our main components have in our model?

To evaluate the impact of each main component in our model, we conduct an ablation study by comparing the performance of two individual models and their final fusion model. The results of the two models in Table II show that the R-GCN model performs better on code smells like *Broken Hierarchy* and *Insufficient Modularization*, while the LSTM model performs better on code smells like *Deficient Encapsulation* and *Insufficient Modularization*. However, each model has its own limitations and neither of them can effectively capture all the features of comprehensive features of code smells. Meanwhile, R-GCN does not work well for *Deficient Encapsulation* because *Deficient Encapsulation* focuses more on the encapsulated information inside class members, so the LSTM using AST tokens can better extract the details of the class members, but this still requires information from the class relation to determine if these class members should be encapsulated through inter-class relation.

C. RQ3:How does the class-diagram contribute to our proposed model?

In Section III, we introduce the class diagram consisting of two parts: extracting complex relation between classes and extracting internal information of each class. To investigate their impact on our DSIR method, we conduct the following experiments. First, we evaluate the impact of different types

of relation between classes by comparing the performance of a Graph Convolutional Network model trained on the same dataset but not distinguishing between edge types. Then, we evaluate the impact of class information in the class diagram by removing method and attribute information from the previously mentioned tuples and only using the class name as a feature.

As shown in Table II, different types of relation between classes can effectively help the model learn the external hierarchy of classes and the interactions between modules. The experiment without relation types results in a 28% decrease in F1 score for *Broken Hierarchy* and a 47% decrease in F1 score for *Insufficient Modularization*. However, for *Deficient Encapsulation*, the relation types do not have a significant effect, as this smell focuses more on the encapsulation information of class members. In this case, class methods and attributes are more useful for understanding the internal features of classes in complex relations, providing encapsulation information and improving efficiency. Moreover, the information of methods and attributes as class internal features can also help the model better understand issues such as function confusion or complex responsibilities within classes, which are associated with *Broken Hierarchy* and *Insufficient Modularization*.

In summary, the inclusion of different types of class relation and internal information of each class in class diagrams enhances the model's discriminative and expressive capabilities, thus improving the performance of code smell detection.

VI. Threats to validity

A. Internal validity

We utilize the UMLGraph tool to construct a class diagram dataset and remove projects where class diagrams could not be automatically generated due to tooling issues. This may introduce bias if these projects have different characteristics than those in our dataset. Meanwhile, we use the Designite tool to label our training data as ground truth. Although the tools are widely used, their reliability still needs to be verified.

B. External validity

We only use Java projects from GitHub, which may limit our ability to extend our findings to other programming languages or domains. Additionally, the limited number of positive and negative samples and the random reduction of negative samples to match the positive samples may introduce sampling bias if the sampled projects have different characteristics than the population of Java projects.

VII. Conclusion

In this paper, we propose a new approach for detecting design smells, namely Design Smell Detection through Inter-class Relation. We apply the AST node tokens instead of code tokens or AST, and inter-class relation to design smell detection, which are separately fed to BiLSTM and R-GCN networks to extract the comprehensive features. Extensive experiments are conducted on a new dataset, and the results demonstrate that the proposed method outperforms state-of-the-art methods in terms of F1-measure of all code smells.

A cknowledgment

This work is partially supported by the National Natural Science Foundation of China (62172202), Collaborative Innovation Center of Novel Software Technology and Industrialization, the Major Program of the Natural Science Foundation of Jiangsu Higher Education Institutions of China under Grant Nos. 22KJA520008, the Priority Academic Program Development of Jiangsu Higher Education Institutions, and the Undergraduate Training Program for Innovation and Entrepreneurship, Soochow University(202210285193H).

References

[1] M. Fowler, *Refactoring - Improving the Design of Existing Code*, ser. Addison Wesley object technology series. Addison-Wesley, 1999. [Online]. Available: http://martinfowler.com/books/refactoring.html

[2] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.

[3] A. M. Fard and A. Mesbah, "Jsnose: Detecting javascript code smells," in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2013, pp. 116–125.

[4] M. Grandini, E. Bagli, and G. Visani, "Metrics for multi-class classification: an overview," *arXiv preprint arXiv:2008.05756*, 2020.

[5] K. Gupta, D. Song, and K. Sen, "Neural-based heuristic search for program synthesis," in *EECS-2020-135*, 2020.

[6] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

[7] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone smells in software evolution," in *2007 IEEE International Conference on Software Maintenance*, 2007, pp. 24–33.

[8] A. Kaur and S. Singh, "Detecting software bad smells from software design patterns using machine learning algorithms," 2018.

[9] T. Sharma, F. Palomba, and D. Spinellis, "On the feasibility of transfer-learning code smells using deep learning," *arXiv preprint arXiv:1904.03031*, 2019.

[10] T. Guggulothu and S. A. Moiz, "Code smell detection using multi-label classification approach," *Software Quality Journal*, vol. 28, no. 3, pp. 1063–1086, 2020.

[11] W. Xu and X. Zhang, "Multi-granularity code smell detection using deep learning method based on abstract syntax tree," in *Proceedings of the 33rd International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 07 2021, pp. 503–509.

[12] Y. Li, A. Liu, L. Zhao, and X. Zhang, "Hybrid model with multi-level code representation for multi-label code smell detection (077)," *International Journal of Software Engineering and Knowledge Engineering*, pp. 1–24, 2022.

[13] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, "Deep learning based code smell detection," *IEEE transactions on Software Engineering*, pp. 1811–1837, 2019.

[14] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[15] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 255–258.

[16] P. Chen and W.-T. Zhang, "Uml-based design and analysis of web applications," *Journal of Systems and Software*, vol. 66, no. 3, pp. 189–200, 2003.

[17] G. Booch, J. Rumbaugh, and I. Jacobson, "The unified modeling language user guide," in *Addison-Wesley Professional*, 2005.

[18] D. Kang, B. Xu, J. Lu, and W. Chu, "A complexity measure for ontology based on uml," in *Proceedings. 10th IEEE International Workshop on Future Trends of Distributed Computing Systems, 2004. FTDCS 2004.*, 2004, pp. 222–228.

[19] T. Sharma, P. Mishra, and R. Tiwari, "Designite: a software design quality assessment tool," in *Bridge*, 2016.

[20] A. Hamdy and M. Tazy, "Deep hybrid features for code smells detection," *Journal of Theoretical and Applied Information Technology*, vol. 98, pp. 2684–2696, 07 2020.