计算机科学
COMPUTER SCIENCE

支持类型敏感场景的跨过程代码依恋检测

厉剑豪, 白瑶瑶, 密杰, 张迎周, 曹文龙, 王栋, 王刚

引用本文

厉剑豪, 白瑶瑶, 密杰, 张迎周, 曹文龙, 王栋, 王刚. 支持类型敏感场景的跨过程代码依恋检测[J]. 计算机科学, 2025, 52(12): 32-39.

LI Jianhao, BAI Yaoyao, MI Jie, ZHANG Yingzhou, CAO Wenlong, WANG Dong, WANG Gang. Cross-procedure Feature Envy Detection Supporting Type-sensitive Scenarios [J]. Computer Science, 2025, 52(12): 32-39.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

基于深度学习的海洋热点新闻挖掘方法

Deep Learning-based Method for Mining Ocean Hot Spot News

计算机科学, 2024, 51(11A): 231200005-10. https://doi.org/10.11896/jsjkx.231200005

基于符号执行优化的PDF恶意指标提取技术

PDF Malicious Indicators Extraction Technique Based on Improved Symbolic Execution

计算机科学, 2024, 51(7): 389-396. https://doi.org/10.11896/jsjkx.230300117

面向JavaScript引擎报错机制的类别导向模糊测试方法

Category-directed Fuzzing Test Method for Error Reporting Mechanism in JavaScript Engines

计算机科学, 2023, 50(12): 49-57. https://doi.org/10.11896/jsjkx.221200166

过程间流敏感的指针分析技术研究

Survey of Interprocedural Flow-sensitive Pointer Analysis Technology

计算机科学, 2023, 50(12): 1-13. https://doi.org/10.11896/jsjkx.221000195

基于注意力机制的多模态在线评论有用性预测研究

Study on Multimodal Online Reviews Helpfulness Prediction Based on Attention Mechanism

计算机科学, 2023, 50(8): 37-44. https://doi.org/10.11896/jsjkx.220600204

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.241200007

支持类型敏感场景的跨过程代码依恋检测

厉剑豪　白瑶瑶　密　杰　张迎周　曹文龙　王　栋　王　刚

南京邮电大学计算机学院　南京 210023

(lijh0410@163.com)

摘　要　代码依恋现象的存在会影响系统的稳定性和可维护性。目前的代码依恋检测方法均未考虑对象类型的敏感性,导致检测精度较低。为解决此问题,提出一种基于高阶函数的过程间代码依恋检测方法。该方法根据预定义的代码依恋度量规则,将过程内带参数的局部性引用比的计算过程抽象为可复用的高阶函数式摘要;过程间检测时,在方法调用点处取出目标方法的高阶代码依恋检测摘要,并根据形实参对应关系将形参的实际类型代入摘要中,计算得到最终的局部性引用比集合,基于该集合来检测代码依恋现象以及对应的依恋集。整合了部分 Java 项目作为基准测试集,选取 IntelliJDeodorant 和 IDE Inspection 工具进行对比实验,结果表明:提出的方法在检测依恋实例的精度上较 IDE Inspection 提高了 16.6%,比 IntelliJDeodorant 提高了 1.3 倍;在检测依恋集的精度上较 IDE Inspection 提高了 37.2%,比 IntelliJDeodorant 提高了 1.6 倍。

关键词:代码依恋检测;类型敏感;过程间;高阶函数;函数摘要;Java

中图分类号　TP311

Cross-procedure Feature Envy Detection Supporting Type-sensitive Scenarios

LI Jianhao, BAI Yaoyao, MI Jie, ZHANG Yingzhou, CAO Wenlong, WANG Dong and WANG Gang

School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China

Abstract　The existence of feature envy phenomena can affect the stability and maintainability of the system. Current feature envy detection methods fail to consider the sensitivity of object types, leading to low detection accuracy. To address this issue, an inter-procedure feature envy detection method based on higher-order function is proposed. The method stores the computation process of the local reference ratios within a procedure with parameters as a summary in the form of higher-order function, according to predefined feature envy measurement rules. During inter-procedure detection, it retrieves the higher-order feature envy detection summary of the target method at the method call site. Based on the correlation between formal and actual parameters, the actual types of the formal parameters are substituted into the summary to compute the final local reference ratios set, which is used to detect the presence of feature envy and corresponding envy sets. The paper integrates several Java projects as a benchmark test set and selects the IntelliJDeodorant and IDE Inspection tools for comparison experiments. Results show that the proposed method improves detection accuracy for envy instances by 16.6% over IDE Inspection and by 1.3 times over IntelliJDeodorant. In terms of envy sets detection accuracy, it improves by 37.2% over IDE Inspection and by 1.6 times over IntelliJDeodorant.

Keywords　Feature envy detection, Type-sensitive, Inter-procedure, High-order function, Function summary, Java

1　引言

Fowler 等[1]将软件开发过程中引入的不良设计或者不合适的编码称为代码异味(Code Smell)。代码依恋(Feature Envy)是一种典型且常见的代码异味,它是指一个类过度依赖另一个类中的属性或方法,而非利用自身的属性或方法来实现功能的现象。这种现象的存在会增加代码模块之间的耦合度,从而影响软件项目的稳定性、可维护性和可扩展性。为了确保软件项目的健康发展,识别并消除系统中的代码依恋现象具有重要意义。目前,代码依恋检测方法已得到广泛研究,大多数检测方法在设立依恋度量指标时都会涉及方法所在类和目标类,但均未考虑到在类型敏感[2]环境下目标类的类型可能会发生改变,这将影响相关指标的计算,从而影响代码依恋检测的精度。

图 1 展示了一个类型敏感环境下涉及多态属性的 Java 程序。类 B 中的方法 t( )显然是存在代码依恋现象的,依恋类为 View,但是通过过程间代码依恋检测可以发现,在方法 p( )中调用方法 t( )时传入的实参 rv 指向的对象类型是 RV,这说明方法 t( )中实际调用的是类 RV 中的属性和方法。按照重构的主观性原则以及依恋度量规则,方法 t( )的

到稿日期:2024-12-02　返修日期:2025-03-28

基金项目:国家自然科学基金(62272214)

This work was supported by the National Natural Science Foundation of China(62272214).

通信作者:张迎周(zhangyz@njupt.edu.cn)

最终依恋类应该是 RV，这就是在类型敏感环境下上下文调用时对代码依恋类产生的影响。

```
public class View {
        inti;
        int j;
        private int m() {
            returni * j;
        }
}
public class RV extends View {
}
public class B {
        private int t(View v) {
                returnv. m()+v. j+v. i;
        }
        private int p() {
                Viewrv=new RV();
                return t(RV);
        }
}
```

图 1　Java 程序示例

Fig. 1　Java program example

为了提高在类型敏感环境下代码依恋检测的精度，本文提出了基于高阶函数的过程间代码依恋检测方法。本文的主要工作如下：

1)在进行过程内代码依恋检测时，用前向数据流分析技术进行类型敏感分析，搜集方法内每个程序点处的对象引用变量的类型信息，同时基于该类型信息更新外部类和内部类的访问频率；数据流迭代结束之后，对最终的外部类访问频率集合和内部类访问频率集合进行计算，得到方法内每个外部类的局部性引用比，并将该计算过程抽象为可复用的高阶函数[3-4]式摘要。

2)在进行过程间代码依恋检测时，在方法调用点处，根据形实参对应关系将形参的实际类型代入高阶代码依恋检测摘要中计算，即可得到真实的局部性引用比集合，遍历该集合来检测方法内部是否存在代码依恋现象以及对应的依恋集。

3)通过多组对比实验，从代码依恋实例的检测精度、代码依恋集的检测精度、方法通用性 3 个方面验证了所提方法的有效性。

2　相关工作

2.1　研究现状

目前的代码依恋检测方法可分为基于启发式的检测方法和基于学习的检测方法。

基于启发式的代码依恋检测方法通常会预先设置一系列度量规则以及阈值，在检测过程中一旦设定的阈值被超过，系统就会提示潜在代码依恋现象。广泛使用的代码依恋检测工具 JDeodorant[5] 通过计算方法与自身类和其他类之间的距离来检测代码依恋现象。Sales 等[6] 通过计算方法的依赖集与当前类和其他类中方法的依赖集之间的相似度来检测代码依恋现象。Liu 等[7] 通过构建抽象语法树并计算方法与类之间的依恋度来进行代码依恋检测。Chen 等[8] 利用数据流分析记录方法中每条语句访问的变量及其来源，认为当外部类成员访问次数超过自身类成员时，该方法存在代码依恋现象。但是，该方法实现较为简单，并没有考虑到变量类型可能因为方法内某些语句的执行而改变，或过程间调用时方法的形参类型可能会改变。基于启发式的方法的优点在于实现简单，资源开销小，但也存在一定的局限性，主要体现在人为设定规则的主观性上。

近年来，基于机器学习和深度学习的代码依恋检测方法被广泛研究。Skipina 等[9] 分别使用从开源 Java 项目中提取的代码度量指标和 CodeT5 等嵌入模型生成代码的向量表示训练机器学习模型，来进行代码依恋检测。Priyambadha 等[10] 从类图中提取结构性信息和语义信息训练分类器，来检测代码依恋现象。Liu 等[11] 通过启发式规则和决策树分类器构建了高质量的训练数据，从中提取文本特征和结构特征训练深度学习模型，来预测方法是否应该移动到目标类。Al-Fraihat 等[12] 从数据集中提取 44 个度量指标作为特征来训练多个模型检测代码依恋，其中 WeightedEnsemble_L2 模型表现最佳。Yu 等[13] 先将项目中的方法调用关系和代码度量信息转换为图的节点和边，然后使用 GNN 分类器对图进行训练，以预测存在代码依恋的方法。基于学习的检测方法通常能取得较好的效果，但大多数方法在初期仍依赖启发式规则来生成或标注训练数据[14-15]。因此，本文提出的代码依恋检测方法也是基于启发式的。

2.2　符号解释

本文使用类型系统相关表示方法来描述基于高阶函数的过程间代码依恋检测方法中一些关键步骤的分析过程，类型系统部分符号及说明如表 1 所列。

表 1　符号说明

Table 1　Description of symbols

符号/表达式	描述
p,q	对象引用变量
t	形参的实际类型
$Dict$	映射关系的集合类型
σ	在每一个程序点处的数据流值
IN_s	流入每个程序点处的数据流值，初始为空集
$KILL_s$	由于该条程序语句的执行在该程序点处无效化的数据流信息的集合，初始为空集
$m\#p$,$m\#q$	来自两个不同分支的两个同类型变量的值，用#进行区分
$PhiStmt$	在 $Jimple$ 中以 go 语句为主，为了体现通用性而直接以 $\Phi(p,q)$ 的形式命名
q=virtualinvoke $p.\langle T:M\rangle()$	当前方法访问外部类 T 中的方法 M
q=specialinvoke this. $\langle T:M\rangle()$	当前方法访问内部类 T 中的方法 M
$q=t.\langle T:F\rangle$	当前方法访问类 T 的属性 F。当 $t=this$ 时，该语句访问当前类内的属性，否则为访问外部类的属性
σ_{of}	当前方法访问的外部类集合
σ_{if}	当前方法访问的内部类集合
snd	取出对当前对象引用变量的访问频率数值
$updateSnd$	在迭代的过程中更新当前引用对象变量的访问频率数值
val	取出当前内部类的访问频率数值
$updateVal$	在迭代过程中更新当前内部类的访问频率数值
$updateFst$	根据形实参对应关系，用形参的真实类型实例化外部类访问集合 σ_{of} 中的参数变量
FS	外部类 T_o 和其对应的 LRR 值之间的映射关系

3　基于高阶函数的过程间代码依恋检测

现阶段,代码依恋检测方法的分析范围仅局限于单个方法内部,分析的度量指标及发现的特征也仅仅局限于单个类。而在面向对象程序中,多态属性的存在使得在上下文调用时被调用方法的形参类型可能会发生改变,这不仅会影响代码依恋检测的精度,也会改变依恋集的内容,从而降低代码重构的准确性。

因此,本章提出了基于高阶函数的过程间代码依恋检测方法。该方法将形参的类型参数化,将带参数的局部性引用比的计算过程抽象为可复用的高阶函数式摘要,在方法调用点处按照形实参对应关系代入形参的实际类型进行计算,得到了真实的局部性引用比集合,遍历该集合并结合依恋阈值进行代码依恋检测。整个工作流程如图 2 所示。

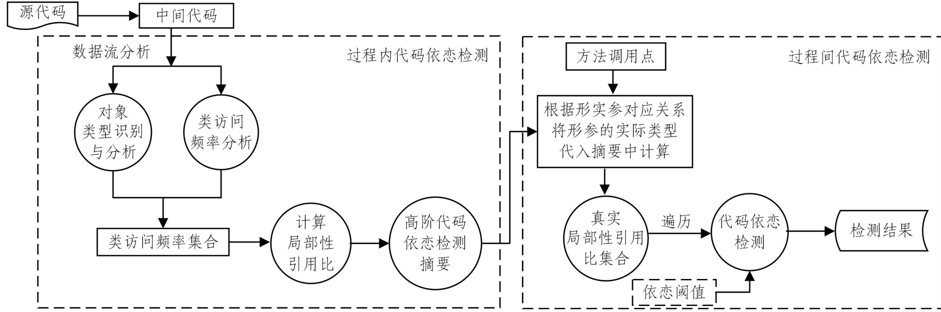

图 2　总体框架

Fig.2　Overall framework

3.1　代码依恋度量

结合文献[7]中所总结的代码依恋规则,定义了以下几类代码依恋度量。

定义 1(内部类访问频率,Self Access Frequency,SAF)　内部类访问频率指的是方法 M 访问其所在类 C 中的成员属性或者方法的次数,用 $SAF_{M,C}$ 表示。

定义 2(外部类访问频率,External Access Frequency,EAF)　外部类访问频率指的是方法 M 访问某个外部类 C' 中的成员属性或者方法的次数,用 $EAF_{M,C'}$ 表示。

定义 3(局部性引用比,Locality Reference Ratio,LRR)　对于类 C 中的方法 M,局部性引用比指的是外部类访问频率与内部类访问频率的比值,用 $LRR_{M,C,C'}$ 表示。$LRR_{M,C,C'}$ 的计算式如下:

$$LRR_{M,C,C'}=\frac{\sum\limits_{\forall C'\neq C} EAF_{M,C'}}{SAF_{M,C}} \tag{1}$$

式(1)的外部类具有唯一性,不具备叠加性。如果方法 M 调用了一组外部类 $\{C_1,C_2,\cdots,C_n\}$,可以定义一个映射 LRR_M,它将每个外部类 C_i 映射到其对应的局部性引用比。LRR_M 表示为:

$$LRR_M:\{C_1,C_2,\cdots,C_n\}\rightarrow R \tag{2}$$

在实际应用中,需要遍历这个映射来分析和比较方法 M 对不同外部类的依恋程度。

定义 4(依恋阈值)　依恋阈值定义了局部性引用比的临界点。在本文中,对于类 C 中的方法 M,如果存在某个外部类 C',使得 $EAF_{M,C'}>SAF_{M,C}$,即 $LRR_{M,C,C'}>1$,则判断类 C 中的方法 M 对外部类 C' 存在代码依恋现象。

定义 5(依恋集,Affinity Set,AS)　对于类 C 中的方法 M,其依恋集定义为方法 M 所依恋的外部类的集合,用 AS_M 来表示:

$$AS_M=\{Ci\mid LRR_{M,C,Ci}>1,\forall i\in\{1,2,\cdots,n\}\} \tag{3}$$

本文将依恋集定义为代码依恋检测中的重要一环,通过引入依恋集的概念,能更系统地识别和处理代码中的依赖问题,也能为代码重构[16]提供更可靠的保障。

3.2　过程内代码依恋检测

3.2.1　过程内对象类型识别与分析

本文以 Java 语言为例,将程序源代码编译生成基于 Jimple 的中间代码表示(IR),通过分析 IR 得到程序的函数调用图(CG)和方法内控制流图(CFG),并在 CFG 上利用前向数据流分析算法进行流敏感的过程内对象类型分析。本文在 IR 上定义了几类影响依恋类型的语句,并用类型系统描述处于这几类程序点时数据流值的分析过程。具体的分析规则如表 2 所列,其中的 Dict[Str,Str]表示对象引用变量及其类型之间的映射关系。

表 2　对象类型分析规则

Table 2　Object types analysis rules

Stmt and Example	Rule
IdentityStmt p=T	$\dfrac{\Gamma\vdash p:T\ \ \Gamma\vdash T:Str\ \ \Gamma\vdash IN_s:Dict[Str,Str]\ \ \Gamma\vdash s:Stmt\ \ \Gamma\vdash \sigma:Dict[Str,Str]\ \ \Gamma,\text{type }Map=Dict[Str,Str]}{\Gamma\vdash \lambda s.[(\sigma=\{p\mapsto T\})\cup(IN_s-KILL_s)]:Stmt\rightarrow Map}$ $\Gamma\vdash KILL_s:Map$
AssignStmt p=q	$\dfrac{\Gamma\vdash p:Str\ \ \Gamma\vdash q:Str\ \ \Gamma\vdash IN_s:Map\ \ \Gamma\vdash \sigma:Map\ \ \Gamma\vdash \sigma[q]:Str\ \ \Gamma\vdash s:Stmt}{\Gamma\vdash \lambda s.[(\sigma=\{p\mapsto \sigma[q]\})\cup(IN_s-KILL_s)]:Stmt\rightarrow Map}$ $\Gamma\vdash KILL_s:Map$
NewStmt p=new T	$\dfrac{\Gamma\vdash p:T\ \ \Gamma\vdash T:Str\ \ \Gamma\vdash IN_s:Map\ \ \Gamma\vdash \sigma:Map\ \ \Gamma\vdash s:Stmt}{\Gamma\vdash \lambda s.[(\sigma=\{p\mapsto T\})\cup(IN_s\text{-}KILL_s)]:Stmt\rightarrow Map}$ $\Gamma\vdash KILL_s:Map$
PhiStmt m=Φ(p,q)	$\dfrac{\Gamma\vdash p:Str\ \ \Gamma\vdash q:Str\ \ \Gamma\vdash m:Str\ \ \Gamma\vdash IN_s:Map\ \ \Gamma\vdash \sigma:Map\ \ \Gamma\vdash \sigma[p]:Str\ \ \Gamma\vdash \sigma[q]:Str\ \ \Gamma\vdash s:Stmt}{\Gamma\vdash \lambda s.[(\sigma=\{m\#p\mapsto \sigma[p]\}+\{m\#q\mapsto \sigma[q]\})\cup(IN_s-KILL_s)]:Stmt\rightarrow Map}$ $\Gamma\vdash KILL_s:Map$

结合表 2 中的推断规则,沿着 CFG 图进行数据流迭代分析,最终可以得到方法内每个程序点处的对象引用变量和其类型之间映射关系的集合。具体的对象类型分析算法如算法 1 所示。

算法 1 流敏感对象类型分析算法

输入：方法控制流图 CFG

输出：方法内所有对象引用变量和其对象类型之间的映射关系集合 Dict

1. defdataFlowSensitiveTypeAnalysis(CFG)：
2. WorkList← Ø，IN={}，OUT={}，Dict={}；
3. for each block in CFG. Blocks do //初始化每个程序点处的数据流值
4. foreachn in block. Instructions do
5. IN[n]={}；
6. OUT[n]={}；
7. end for
8. end for
9. WorkList. add(CFG. ENTRY)；//CFG 的入口节点，这里把每条指令作为一个节点
10. while WorkList is not empty do //workList 算法
11. node ←WorkList. pop()；
12. for eachpre_node in $node_{predecessors}$ do // 遍历 node 节点的所有前驱
13. merge(IN[node]，OUT[pre_node]) //合并 node 节点的所有前驱节点的 OUT 值，得到流入新节点的 IN 值
14. end for
15. GEN_{node}← TypeAnalysis(node)；//依照表 2 的对象类型分析规则搜集对象变量及其类型信息，GEN_{node}的类型依然为 Dict[Str，Str]
16. NEW_OUT_{node}← GEN_{node}∪(IN[node]－$Kill_{node}$)；
17. if OUT[node] ≠ NEW_OUT_{node} then
18. WorkList. add($node_{successors}$)；//添加 node 节点的所有后继节点
19. OUT[node] ← NEW_OUT_{node}
20. end if
21. end while
22. Dict ← NEW_OUT_{node}；
23. returnDict

3.2.2 高阶函数式类访问频率分析

由于在进行数据流迭代的过程中还需要记录内部类访问频率 *SAF* 和外部类访问频率 *EAF* 的数值，因此本节对 IR 中与计算 *SAF* 和 *EAF* 相关的语句进行分析。具体的分析规则如表 3 所列。

表 3 EAF 和 SAF 分析规则

Table 3 EAF and SAF analysis rules

Stmt	Rule
q=virtualinvoke p.⟨T:M⟩() / q=p.⟨T:F⟩	$\dfrac{\Gamma\vdash a:A\ \ \Gamma\vdash b:B}{\Gamma\vdash(a,b):Tuple[A,B]}$ $\Gamma\vdash p:Str\quad\Gamma\vdash T:Str\quad\Gamma\vdash\sigma:Dict[Str,Str]$ $\dfrac{\Gamma\vdash\sigma_{of}:Dict[Str,TS]\quad\Gamma\vdash TS:Tuple[Str,Int]}{\Gamma\vdash if\ \sigma_{of}[p]\rightarrow\emptyset then\ \sigma_{of}'=\sigma_{of}+\{p\mapsto(\sigma[p],1)\}}$ $else\ updateSnd(p,snd(\sigma_{of}[p])+1):Dict[Str,TS]$ $\Gamma\vdash\sigma_{of}[p]:TS\quad\Gamma\vdash\sigma_{of}':Dict[Str,TS]$
q=specialinvoke this.⟨T:M⟩() / q=this.⟨T:F⟩	$\dfrac{\Gamma\vdash\sigma_{if}:Dict[Str,Int]\quad\Gamma\vdash T:Str}{\Gamma\vdash if\ \sigma_{if}[T]\rightarrow\emptyset\ then\ \sigma_{if}'=\sigma_{if}+\{T\mapsto1\}}$ $elseupdateVal(\sigma_{if}[T],val(\sigma_{if}[T])+1):Dict[Str,Int]$ $\Gamma\vdash\sigma_{if}[T]:Int\quad\Gamma\vdash\sigma_{if}':Dict[Str,Int]$

表 3 中的 Dict[Str，Str]表示对象引用变量及其类型之间的映射关系；Dict[Str，Ts]表示对象引用变量和二元组之间的映射关系；Tuple[Str，Int]表示对象引用变量在当前类内的类型名称以及当前对象引用变量的访问频率；Dict[Str，Int]表示内部类的类型名和访问频率的映射关系。

当数据流迭代分析到达不动点时，得到包含参数变量在内的外部类访问频率集合 σ_{of} 和内部类访问频率集合 σ_{if}。由于 σ_{if} 中的对象类型为方法 *M* 的所在类，因此遍历集合 σ_{if} 可以确定 *SAF* 的值。而 σ_{of} 中包含多个外部类以及参数，根据定义 2 计算每个外部类的访问频率值 *EAF*(*To*)，再依照定义 3 计算每个外部类的局部性引用比 *LRR*(*To*)。具体规则如式(4)所示，*p* 表示任意一个类型为 *To* 的对象引用变量。

$$\begin{aligned}&EAF(T_o)=sum(\{n\mid(p\rightarrow(T_o\rightarrow n))\in\sigma_{of}\})\\&SAF=sum(\{n\mid(T_i\rightarrow n)\in\sigma_{if}\})\\&LRR(T_o)=EAF(T_o)/SAF\end{aligned}\tag{4}$$

由于 σ_{of} 中包含参数，在多态属性的作用下，过程间方法调用时可能导致外部类的类型 *To* 发生改变，这将影响 *EAF*(*To*)的计算。本文利用高阶函数的“函数可作为返回值”的特性，将 *LRR* 的计算过程抽象为可复用的高阶函数式摘要并返回。高阶代码依恋检测摘要的构建规则如式(5)所示。这里以一个形参 *q* 为例，用字符串类型的参数 *v* 表示该形参的类型，*t* 表示该形参的实际类型并且由方法被调用时的上下文信息决定。过程内代码依恋检测的全部实现过程如算法 2 所示。

$$\frac{\begin{array}{c}\Gamma\vdash t:Str\Gamma\vdash q:v\Gamma\vdash v:Str\\ \Gamma\vdash\sigma_{of}:Dict[Str,TS]\Gamma\vdash\sigma_{if}:Dict[Str,Int]\\ \Gamma\vdash FS:Dict[Str,Int]\end{array}}{\begin{array}{c}\Gamma\vdash\lambda t.\{updateFst(\sigma_{of},v,t);\\ FS+\{To\mapsto(sum(\{n\mid(p\rightarrow(To,n))\in\sigma_{of}\})/\\ sum(\{n\mid(T_i\rightarrow n)\in\sigma_{if}\}))\}:Str\rightarrow Dict[Str,Int]\end{array}}\tag{5}$$

算法 2 过程内代码依恋检测算法

输入：待分析函数的控制流图 CFG

输出：表示分析计算 LRR 的过程高阶函数 analysisLRR

1. def intralFeatureEnvyAnalysis(CFG)：
2. σ_{of}← {}，σ_{if}← {}，σ ← {}，EAF ← {}，SAF ← 0，FS ← {}；
3. σ_{upof}← {}；//定义参数更新之后的外部类访问频率集合
4. paramVarList← exFormalParamType(paramList)//将形参的类型按序用字符串类型的参数变量表示
5. foreach block in CFG. Blocks do
6. foreach inst in Block. Instructions do
7. …//省略算法 1 流敏感对象类型分析算法框架
8. σ ←TypeAnalysis(inst)；//基于算法 1 得到该程序点处的数据流值 σ
//在 TypeAnalysis 中加入额外的类访问频率分析，根据 EAF 和 SAF 的分析规则，只针对相关语句分析，部分指令用正则表达式匹配
9. If inst==virtualinvoke ||inst. match(“\\w+\\.⟨\\w+:\\s\\w+\\s\\w+⟩”) then
10. p ←getObjectVar(inst)；//得到调用类的实例化变量
11. T_o← σ[p]；//得到调用的外部类类型

12. if $\sigma_{of}[p] \leftarrow \emptyset$ then
13. $\sigma_{of}[p] \leftarrow (T_o, 1)$;
14. else
15. TS $\leftarrow \sigma_{of}[p]$;
16. count $\leftarrow$ snd(TS)+1;//snd 方法为取出二元组 TS 中的第二个元素的值
17. $\sigma_{of}[p] \leftarrow (T_o, \text{count})$;
18. endif
19. endif
20. if inst == specialinvoke || inst. contains("this\\.") then
21. $T_i \leftarrow$ getClassType(inst);//得到调用的内部类类型
22. if $\sigma_{if}[T_i] \leftarrow \emptyset$ then
23. $\sigma_{if}[T_i] \leftarrow 1$;
24. else
25. count $\leftarrow \sigma_{if}[T_i]+1$;
26. $\sigma_{if}[T_i] \leftarrow$ count;
27. endif
28. endif
29. end for
30. end for
31. def analysisLRR(x):
32. updateFst(σ_{of}, paramVarList, x);//遍历 σ_{of},根据形实参对应关系将参数变量替换为集合 x 中形参的实际类型
33. SAF $\leftarrow \sigma_{if}[T_i]$;//内部类不变,可以直接求得 SAF 的值
34. foreach tuple in σ_{of}. values do
35. T $\leftarrow$ fst(tuple);count $\leftarrow$ snd(tuple);
36. if T is not in σ_{upof} then
37. $\sigma_{upof}[T] \leftarrow$ count;
38. else
39. tempC $\leftarrow \sigma_{upof}[T]$;
40. $\sigma_{upof}[T] \leftarrow$ tempC+count;
41. end if
42. end for
43. foreach T_o in σ_{upof}. keys do
44. FS$[T_o] \leftarrow (\sigma_{upof}[T_o]$ / SAF);//根据定义 3 计算 LRR,只有当前计算的 $\sigma_{upof}[T_o]$/SAF 大于原本集合中 FS$[T_o]$值时才更新
45. end for
46. returnanalysisLRR;//将分析 LRR 计算过程的函数 analysisLRR作为高阶函数 intralFeatureEnvyAnalysis 的返回值返回

3.3 过程间代码依恋检测

因为 σ_{of} 中可能存在参数,所以需要进一步对方法调用点处的上下文信息进行分析。表 4 列出了在方法调用点处的处理规则,在调用点处取出目标方法已经分析好的高阶代码依恋检测摘要,按照形实参对应关系,将相应的形参的实际类型代入摘要中进行计算,得到方法 *M* 更新后的 *LRR* 集合 *FS*,这里 *r* 表示根据上下文信息得到形参 *q* 的真实类型。具体的过程间代码依恋检测算法如算法 3 所示。

表 4 方法调用点处的处理规则

Table 4 Processing rule at the method call site

Stmt	Rule
virtualinvoke k. $\langle T:M \rangle (q)$	$\dfrac{\begin{array}{l}\Gamma \vdash \sigma_{if}:\text{Dict}[\text{Str},\text{Int}]\ \Gamma \vdash FS:\text{Dict}[\text{Str},\text{Int}] \\ \Gamma \vdash t:\text{Str}\quad \Gamma \vdash r:\text{Str}\ \Gamma \vdash q:v\quad \Gamma \vdash v:\text{Str} \\ \Gamma \vdash \sigma_{of}:\text{Dict}[\text{Str},TS] \\ \Gamma \vdash \lambda t.\{\text{updateFst}(\sigma_{of},v,t); \\ FS+\{T_o \mapsto (\text{sum}(\{n \mid (p \rightarrow (T_o,n)) \in \sigma_{of}\}) / \\ \text{sum}(\{n \mid (T_i \rightarrow n) \in \sigma_{if}\})))\}:\text{Str} \rightarrow \text{Dict}[\text{Str},\text{Int}]\end{array}}{\begin{array}{l}\Gamma \vdash \text{updateFst}(\sigma_{of},v,r) \\ \Gamma, FS+\{T_o \mapsto (\text{sum}\{n \mid (p \rightarrow (T_o,n)) \in \sigma_{of}\}) / \\ \text{sum}(\{n \mid (T_i \rightarrow n) \in \sigma_{if}\})))\}:\text{Dict}[\text{Str},\text{Int}]\end{array}}$

算法 3 过程间代码依恋检测算法

输入:函数调用图 CG,代码依恋检测摘要信息列表 sum_dict,方法对象引用变量与其类型的映射关系集合 σ

输出:LRR 集合 FS

1. funcVetexSet $\leftarrow$ ReverseTopSort(CG);//逆拓扑排序序列
2. foreach func in funcVetexSet do
3. CFG $\leftarrow$ getFuncCFG(func);//得到方法内控制流图
4. calLRRF $\leftarrow$ intralFeatureEnvyAnalysis(CFG);//过程内代码依恋检测,得到高阶代码依恋检测摘要
5. saveSummary(getMethod(CFG), calLRRF);//摘要存储
6. foreach block in CFG. Blocks do //将过程间方法调用处的处理单独拎出
7. foreach inst in block. Instructions do
8. if inst==virtualinvoke then /* 主要处理方法调用点处的摘要计算工作 */
9. callFName $\leftarrow$ GetCSName(inst);//得到被调方法的方法名
10. paramList $\leftarrow$ GetCSParams(inst);//得到被调方法的形参
11. envy_summary $\leftarrow$ sum_dict[callFName];//方法 callFName 的高阶函数形式的代码依恋检测摘要
12. foreach i in len(paramList) do
13. param_dict [i] $\leftarrow$ σ[paramList [i]];//实参为类型,作一个转化
14. end for
/* 将实参代入高阶函数式摘要 envy_summary 中进行计算,得到更新之后的 LRR 集合 */
15. envy_summary(param_dict);//虽然该函数没有返回值,但是 LRR 集合 FS 是一直伴随着计算在更新的
16. end if
17. end for
18. end for
19. end for

在得到方法 *M* 最终的 *LRR* 集合 *FS* 后,结合定义 4 依恋阈值的概念对 *LRR* 集合进行分析,判断当前方法 *M* 中是否存在代码依恋现象以及对应的依恋集。具体代码依恋检测算法如算法 4 所示。

算法 4 代码依恋检测算法

输入:LRR 集合 FS,当前方法 M

输出:代码依恋检测结果 result

1. result $\leftarrow$ {},list $\leftarrow$ [],featureEnvy $\leftarrow$ FALSE;
2. foreach T_o in FS do
3. lrr $\leftarrow$ FS $[T_o]$;//得到类型 T_o的局部性引用比 lrr;

```
4.    if lrr > 1 then
5.       list.add(T_o);
6.       featureEnvy ← TRUE;//TRUE 表示该方法存在代码依恋现象,目标依恋类之一为 T_o。
7.    end if
8. end for
9. result[M] ←(featureEnvy,list);//featureEnvy 为方法 M 是否存在代码依恋的标志,list 为其依恋集,其中包含多个依恋类
10. return result
```

4 实验及分析

本文基于 SootUp 工具实现了基于高阶函数的过程间代码依恋检测方法的原型系统(以下简称 FEHF)。本章对 FEHF 进行了实验,并将实验结果与现阶段的代码依恋检测工具的检测结果进行对比分析。

4.1 实验数据和环境

本文在开源网站上整合了部分 Java Feature Envy 相关测试程序作为本次实验的基准数据集,并统计了每个项目中与实验有关的基本信息,如表 5 所列。表 5 中前 5 个项目中有关代码依恋的程序均涉及类型敏感环境,后 3 个项目中有关代码依恋的程序均不涉及类型敏感环境。另外,为了更好地利用 Soot 框架对 Java 程序进行分析,将 Java 项目都编译成 jar 包,以便进行后续的代码依恋检测。

表 5 实验数据集

Table 5 Experimental dataset

项目	代码依恋实例	依恋集大小	类数量	方法调用点数量
Exfe	1	1	5	6
Yio	2	2	7	13
Topv	5	4	10	28
Aurea	20	8	12	90
Apex	11	6	9	20
Refcare	28	6	16	56
Griffe	13	8	21	62
Socket	8	5	14	40
Average	11	5	11.75	39.375

本文实验环境配置如下:操作系统为 Mac OS,CPU 为 Intel i5,主频 1.4GHz,内存 16GB,SootUp 版本为 1.2.0,IntelliJ Idea 版本为 Ultimate 2024.1.1。

4.2 检测结果指标

为了更好地对比 FEHF 和现阶段的代码依恋检测工具的检测结果,采用了表 6 中的检测指标。

表 6 检测指标

Table 6 Detection metrics

指标	描述
TP	正确检测到的依恋实例/依恋类的个数
FP	误报的依恋实例/依恋类的个数
FN	漏报的依恋实例/依恋类的个数
Precision	检测准确率,表示正确检测到的个数占检测到的总数的比例
Recall	检测召回率,表示正确检测到的个数占真实个数的比例
F1-Score	检测精度,*F1* 值越大,说明 *Precision* 和 *Recall* 值的平衡越好,代码依恋检测的准确度越高

Precision,Recall,F1-Score 的计算方式分别如式(6)—式(8)所示。

$$Precision=\frac{TP}{TP+FP} \tag{6}$$

$$Recall=\frac{TP}{TP+FN} \tag{7}$$

$$F1=2*\frac{Precision*Recall}{Precision+Recall} \tag{8}$$

4.3 实验结果与分析

为了评估基于高阶函数的过程间代码依恋检测方法的有效性,主要从下面 3 个方面进行对比研究。

1)RQ1:在类型敏感环境下,FEHF 对依恋实例的检测精度是否优于现有的代码依恋检测工具?

2)RQ2:在类型敏感环境下,FEHF 对依恋集的检测精度是否优于现有的代码依恋检测工具?

3)RQ3:在类型不敏感环境下,FEHF 是否具有通用性?

针对上述 3 个问题,本文使用自主实现的原型工具 FEHF、Intellij Idea 中的代码依恋检测工具 IntelliJDeodorant 和 IDE 里 Analyze 工具中的 Inspection 对 Java 基准数据集进行代码依恋检测。各工具检测的结果分别如表 7—表 9 所列。

表 7 FEHF 的检测结果

Table 7 Detection results of FEHF

	Exfe	Yio	Topv	Aurea	Apex	Refcare	Griffe	Socket
报告的代码依恋实例	1	2	6	15	13	20	13	10
TP	1	2	5	15	11	19	12	8
FP	0	0	1	0	2	1	1	2
FN	0	0	0	5	0	9	1	0
报告的依恋集大小	1	2	4	9	6	5	6	5
TP	1	2	4	9	6	5	6	5
FP	0	0	0	0	0	0	0	0
FN	0	0	0	0	0	1	2	0

表 8 IntelliJDeodorant 的检测结果

Table 8 Detection results of IntelliJDeodorant

	Exfe	Yio	Topv	Aurea	Apex	Refcare	Griffe	Socket
报告的代码依恋实例	1	2	1	7	2	7	2	3
TP	1	1	1	7	2	7	2	2
FP	0	1	0	0	0	0	0	1
FN	0	1	4	13	9	21	11	6
报告的依恋集大小	1	1	1	2	1	2	1	3
TP	0	1	1	1	1	1	1	2
FP	1	0	0	1	0	1	0	1
FN	1	1	3	7	5	4	7	3

表 9 IDE Inspection 的检测结果

Table 9 Detection results of IDE Inspection

	Exfe	Yio	Topv	Aurea	Apex	Refcare	Griffe	Socket
报告的代码依恋实例	1	3	5	17	12	14	12	10
TP	1	2	4	15	9	13	10	6
FP	0	1	1	2	3	1	2	4
FN	0	0	1	5	2	15	3	2
报告的依恋集大小	1	2	3	5	3	3	5	3
TP	0	2	3	4	3	2	5	3
FP	1	0	0	1	0	1	0	0
FN	1	0	1	4	3	4	3	2

从表 8 可以看出,虽然 IntelliJDeodorant 检测代码依恋现

象的误报个数较少，但是其漏报个数很多；而从表 7 中可以看出，FEHF 能检测出更多的代码依恋现象。通过分析发现，这一方面这是因为 IntelliJDeodorant 设立的度量规则存在一定的局限性，使其无法检测出一些真实存在的代码依恋现象，例如它无法检测目标依恋类的对象为方法内临时变量的情况，而本文设定的度量规则更加全面通用，适用于更复杂的代码结构；另一方面，在类型敏感环境下，上下文调用时通常会涉及到类型转换，这些复杂的类型信息会导致 IntelliJDeodorant 难以区分方法对其他类的依赖是否合理，从而影响了对方法内代码依恋现象的判断，而本文方法利用数据流分析技术进行类型敏感分析，可以识别方法内每个程序点处的对象引用变量的类型，使用高阶函数摘要可以处理上下文调用时类型转换这些复杂情况。

从表 7 和表 9 可以看出，Inspection 在代码依恋现象检测方面能力较强，但不及本文方法。通过分析发现，Inspection 不仅没有考虑到上下文调用导致的类型变化问题，而且无法处理先后调用同一个对象的属性和方法的行为；而本文提出的 FEHF 不仅可以解决上下文调用带来的类型变化问题，还把对对象的属性和方法的访问行为均认为是访问频率的一次增加操作。另外，从表 7 中可以看出，FEHF 也存在漏报和误报的问题，从本文的基准测试集中可以发现，FEHF 对路径敏感下的代码依恋检测处理得过于保守；此外，如 Refcare 和 Aurea 中涉及到循环类引用的问题，FEHF 无法检测循环类引用下引发的代码依恋现象，所以检测部分 Java 测试集程序中的代码依恋实例时产生了误报和漏报的现象。

为了回答 RQ1，本文统计了各个工具对代码依恋实例检测的精度，并将工具 FEHF 的 $F1$ 值与 IntelliJDeodorant 和 Inspection 的 $F1$ 值进行了对比，结果如表 10 所列。

表 10 各工具检测代码依恋实例的 $F1$ 值

Table 10 $F1$-score statistics for detecting envy instances

	项目	IntelliJDeodorant			IDE Inspection			FEHF				
		P	R	$F1$	P	R	$F1$	P	R	$F1$	较 IJD/%	较 Insp/%
报告代码依恋实例	Exfe	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0	0
	Yio	0.50	0.50	0.50	0.67	0.75	0.86	1.00	1.00	1.00	↑100	↑16.30
	Topv	1.00	0.02	0.33	0.80	0.80	0.80	0.83	1.00	0.91	↑176	↑13.80
	Aurea	1.00	0.35	0.52	0.88	0.75	0.81	1.00	0.75	0.86	↑65	↑6.17
	Apex	1.00	0.18	0.31	0.75	0.82	0.78	0.85	1.00	0.92	↑197	↑18.00
	Refcare	1.00	0.25	0.40	0.93	0.46	0.62	0.95	0.68	0.81	↑103	↑30.60
	Griffe	1.00	0.15	0.26	0.83	0.77	0.80	0.92	0.92	0.92	↑254	↑15.00
	Socket	0.67	0.25	0.36	0.60	0.75	0.67	0.80	1.00	0.89	↑147	↑32.80
	Average	0.89	0.36	0.46	0.79	0.76	0.79	0.92	0.92	0.92	↑130	↑16.60

从表 10 中可以看出，在类型敏感的环境下，就代码依恋现象的检测精度而言，FEHF 较 IntelliJDeodorant 和 Inspection 都有明显的提升。因此，在类型敏感环境下，FEHF 对依恋实例的检测精度是优于现有的代码依恋检测工具的。

为了回答 RQ2，本文统计了各个工具对依恋集检测的精度，并且将工具 FEHF 的 $F1$ 值与 IntelliJDeodorant 和 Inspection 的 $F1$ 值进行了对比，结果如表 11 所列。

表 11 各工具检测代码依恋集的 $F1$ 值

Table 11 $F1$-score statistics for detecting envy set

	项目	IntelliJDeodorant			IDE Inspection			FEHF				
		P	R	$F1$	P	R	$F1$	P	R	$F1$	较 IJD/%	较 Insp/%
报告代码依恋实例	Exfe	0	0	0	0	0	0	1.00	1.00	1.00	↑	↑
	Yio	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0	0
	Topv	1.00	0.33	0.50	1.00	0.75	0.86	1.00	1.00	1.00	↑100	↑16.3
	Aurea	0.50	0.13	0.21	0.80	0.50	0.62	1.00	1.00	1.00	↑376	↑61.3
	Apex	1.00	0.17	0.29	1.00	0.50	0.67	1.00	1.00	1.00	↑245	↑49.3
	Refcare	1.00	0.33	0.50	0.67	0.33	0.44	1.00	0.83	0.91	↑82	↑88.6
	Griffe	1.00	0.13	0.23	1.00	0.63	0.77	1.00	0.75	0.86	↑274	↑11.7
	Socket	0.67	0.40	0.50	1.00	0.60	0.75	1.00	1.00	1.00	↑100	↑33.3
	Average	0.77	0.31	0.40	0.81	0.54	0.60	0.98	0.95	0.96	↑168	↑37.2

从表 11 中发现，FEHF 检测依恋集的精度明显高于 IntelliJDeodorant 和 IDE Inspection，这一方面是因为 FEHF 能正确地检测到更多的代码依恋实例，所以收集到的依恋类更多；另一方面是受到类型敏感的影响，FEHF 在类型敏感环境下检测代码依恋现象的平均精度要比 IntelliJDeodorant 高 107.6%，比 IDE Inspection 高 10.85%（见表 10），而 FEHF 在检测代码依恋集的精度上平均比 IntelliJDeodorant 高 180%，比 Inspection 高 31.73%（见表 11），这意味着表 10 中发现的代码依恋实例的依恋类和真实的依恋类有所差别。比如，IntelliJDeodorant，Inspection 和 FEHF 均检测到了 Exfe 中存在的代码依恋现象，但是在报告依恋类时 IntelliJDeodorant 和 Inspection 均出现了错误。通过分析这个测试程序可以看出，该程序中依恋类是通过过程间方法调用传入的，而多态属性的存在导致实际传入方法中的类型与参数类型并不一致，所以 IntelliJDeodorant 和 Inspection 均认为重构应该 Move To 参数变量类型；而 FEHF 进一步进行过程间分析得到重构应该 Move To 传入方法的实际类型的结论，所以对于表 11 中的 Exfe 程序，IntelliJDeodorant 和 IDE Inspection 的 $F1$ 值为 0，表示没有检测到正确的依恋类，FEHF 的 $F1$ 值为 1，表示找到了正确的目标依恋类。由于无法描述 0 到 1 的

跨越程度，本文在精度比较上用增长箭头表示一个从无到有的过程，在后续的平均值统计中不会将该条数据统计在内。

因此可以得出结论：在类型敏感环境下，FEHF 对依恋集的检测精度是优于现有的代码依恋检测工具的。

针对 RQ3，从表 10 和表 11 中的实验结果可以看出，对于不含类型敏感的 Java 测试集程序，FEHF 仍然可以检测其代码依恋现象和依恋集，且在检测精度上仍优于 IntelliJDeodorant 和 Inspection，验证了在类型不敏感环境下 FEHF 具有通用性。此外，虽然本文的面向对象语言是以 Java 为例的，但本文的分析是在中间语言上进行的，该方法对于其他的面向对象语言同样适用。

结束语 本文对类型敏感环境下跨过程的代码依恋检测方法进行了研究，提出了一种基于高阶函数的过程间代码依恋检测方法。相较于现有的代码依恋检测工具，本文考虑了在类型敏感环境下，对象引用变量的类型可能会因为方法内某些语句的执行而改变，以及过程间方法调用时可能会使形参的类型发生变化这两种情况。本文将过程内带参数的局部性引用比的计算过程抽象为可复用的高阶函数式摘要，进行过程间检测时，在方法调用点处取出目标方法的高阶代码依恋检测摘要，并将实际参数类型代入计算，得到真实的局部性引用比集合，通过遍历该集合，检测代码依恋现象以及对应的依恋集。对比实验证明了，本文方法能有效提高在类型敏感环境下跨过程的代码依恋检测的精度，并且在类型不敏感环境下也具有通用性。

后续的相关工作方向主要有：1）本文方法仍然存在一定的漏报和误报，未来可以在该框架下继续研究如路径敏感环境下的代码依恋检测方法，进一步提高检测的精度；2）从本文的对比实验中可以发现，代码依恋的精度具有一定的主观性，因为这是基于每一个代码依恋检测方法所设立的度量规则进行的，未来需要进一步确立一个大众统一的度量规则。

参考文献

[1] FOWLER M，BECK K，BRANT J，et al. Refactoring：improving the design of existing code［M］. Massachusetts：Addison-Wesley，1999.

[2] TAN T，MA X X，XU C，et al. Overview of Java Pointer Analysis［J］. Journal of Computer Research and Development，2023，60(2)：274-293.

[3] HU Z，HUGHES J，WANG M. How functional programming mattered［J］. National Science Review，2015，2(3)：349-370.

[4] ZHANG Y Z，ZHANG W F. Haskell：A Modern Purely Functional Programming Language［J］. Journal of Nanjing University of Posts and Telecommunications（Natural Science），2007（4）：13-18，23.

[5] FOKAEFS M，TSANTAILS N，CHATZIGEORGIOU A. JDeodorant：Identication and Removal of Feature Envy Bad Smells［C］// Proceedings of the 23rd IEEE International Conference on Software Maintenance. IEEE，2007：519-520.

[6] SALES V，TERRA R，MIRANDA L F，et al. Recommending Move MethodRefactorings Using Dependency Sets［C］// Proceedings of the 20th Working Conference on Reverse Engineering. IEEE，2013：232-241.

[7] LIU D D，ZHAO F Y. Research on feature envy detection and refactoring［J］. Electronic Science and Technology，2016，29(11)：70-73.

[8] CHEN W K，LIU C H，LI B H. A feature envy detection method based on dataflow analysis［C］// IEEE 42nd Annual Computer Software and Applications Conference. IEEE，2018：14-19.

[9] SKIPINA M，SLIVKA J，LUBURIC N，et al. Automaticdetection of Feature Envy and Data Class code smells using machine learning［J］. Expert Systems With Applications，2024，243：122855.

[10] PRIYAMBADHA B，KATAYAMA T，KITA Y，et al. Detection of Blob and Feature Envy Smells in a Class Diagram using Class's Features［J］. Journal of Robotics，Networking and Artificial Life，2022，9(1)：43-48.

[11] LIU B，LIU H，LI G J，et al. Deep Learning Based Feature Envy Detection Boosted by Real-World Examples［C］// Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM，2023：908-920.

[12] AL-FRAIHAT D，SHARRAB Y，AL-GHUWAIRI A R，et al. Detecting and resolving feature envy throughautomatedmachine learning and move method refactoring［J］. International Journal of Electrical & Computer Engineering，2024，4(2)：2330-2343.

[13] YU D J，XU Y H，WENG L H，et al. Efficient feature envy detection and refactoring based ongraphneural network［J］. Automated Software Engineering，2025，32(1).

[14] GRUJIC K G，PROKIC S，KOVACEVIC A，et al. Machine Learning Approaches for Code Smell Detection：A Systematic Literature Review［J/OL］. Social Science Research Network，2022. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4299859.

[15] LEWOWSKI T，MADEYSKI L. Code smells detection using artificial intelligence techniques：A business-driven systematic review［M］// Studies in Systems，Decision and Control. 2021：285-319.

[16] BLOCH J. Effective Java［M］. Beijing：Posts & Telecom Press，2024.

LI Jianhao，born in 2001，postgraduate，is a member of CCF（No. E4926G）. His main research interest is program analysis.

ZHANG Yingzhou，born in 1978，professor，is a member of CCF（No. 16845S）. His main research interests include software formal analysis，program slicing，service computing and functional programming.

（责任编辑：喻藜）