软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(8):2505−2521 [doi: 10.13328/j.cnki.jos.006082]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

检测 JavaScript 类的内聚耦合 Code Smell*

黄子杰[1,2], 陈军华[1], 高建华[1]

[1](上海师范大学 计算机科学与技术系,上海 200234)

[2](华东理工大学 计算机科学与工程系,上海 200237)

通讯作者: 高建华, E-mail: jhgao@shnu.edu.cn

摘 要: Code Smell 是软件程序中存在不良设计和不良实现的征兆.正确地检测和识别 Code Smell 可以指导软件重构,提高软件的可用性和可靠性.通过 Code Smell 的度量指标,可以量化软件的设计问题.JavaScript 已成为最常用的编程语言之一,类是 JavaScript 的设计模式,优秀类的设计体现为高内聚和低耦合.现有关于 JavaScript 内聚耦合的 Code Smell 研究均在微观的层面,即函数和语句上进行.它们可以提供程序实现的重构建议,但无法分析内聚耦合相关的软件系统设计问题.针对 FE、DC 和 Blob 这 3 种类的内聚耦合 Code Smell,提出一种 JavaScript 类的内聚耦合 Code Smell 检测方法 JS4C.该方法基于静态分析,同时适用于客户端和服务端程序.它通过遍历软件系统中所有的类,利用源程序的文本相似度特征和结构特征,识别 Code Smell 并检测其强度.在结构特征检测中,JS4C 使用了经扩展的对象类型推断及非严格的耦合分散度度量法 NSCDISP,有效地降低了解释型语言的静态分析过程中,类型信息缺失对检测产生的影响.实验通过对 6 个开源项目的分析表明,JS4C 对内聚耦合设计问题有良好的检测效果.

关键词: Code Smell;JavaScript;内聚;耦合;类

中图法分类号: TP311

中文引用格式: 黄子杰,陈军华,高建华.检测 JavaScript 类的内聚耦合 Code Smell.软件学报,2021,32(8):2505−2521. http://www.jos.org.cn/1000-9825/6082.htm

英文引用格式: Huang ZJ, Chen JH, Gao JH. Detecting coupling and cohesion Code Smells of JavaScript classes. Ruan Jian Xue Bao/Journal of Software, 2021,32(8):2505−2521 (in Chinese). http://www.jos.org.cn/1000-9825/6082.htm

Detecting Coupling and Cohesion Code Smells of JavaScript Classes

HUANG Zi-Jie[1,2], CHEN Jun-Hua[1], GAO Jian-Hua[1]

[1](Department of Computer Science and Technology, Shanghai Normal University, Shanghai 200234, China)

[2](Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai 200237, China)

Abstract: Code Smells are symptoms of poor design and implementation choices. Detect and identify Code Smell precisely provide guidance on software refactoring, and lead to improvement of software usability and reliability. Design problems of software systems could be quantified through Code Smell metrics. JavaScript has become one of the most widely used programming languages, class is a design pattern of JavaScript, loose coupling and strong cohesion are characteristics of a well-designed class. Prior works measured coupling and cohesion Code Smells of JS programs in lower levels, i.e., function-wide and statement-wide, which were capable for providing refactoring suggestions about basic implementations, but not enough to identify design problems. This paper proposed JS4C, a method to detect coupling and cohesion Code Smells of JS classes including FE, DC and Blob. This method is an approach of static analysis works on both server and client-side applications, it iterates over every class in software system and takes advantage of source code textual patterns. While JS4C detects Code Smells, it also determines intensity for each of them. Missing type information in static analysis is reinforced by extended object type inference and non-strict coupling dispersion (NSCDISP) metric during structural analysis. Experiments made on 6 open-sourced projects indicate that JS4C can correctly detect coupling and cohesion design problems.

* 基金项目: 国家自然科学基金(61672355)
Foundation item: National Natural Science Foundation of China (61672355)
收稿时间: 2018-10-31; 修改时间: 2019-10-28, 2020-04-25; 采用时间: 2020-05-12

Key words: Code Smell; JavaScript; cohesion; coupling; class

JavaScript(简称 JS)是一种使用弱类型和动态类型的解释型编程语言,JS 程序的结构和变量类型在运行时才会明确,因此,大量为 Java 等强类型语言设计的静态分析手段难以奏效,导致其代码质量缺乏工具保障.同时,它还包含诸多增强语言灵活度(flexibility)的特性,例如原型链、闭包、头等函数等,它们形成了复杂的代码组件继承、组合方式.因此,编写 JS 程序颇具挑战性,期间易产生 Code Smell 和 Bug[1,2].

Code Smell 是软件程序中存在不良设计和不良实现的征兆[3],其强度可由度量指标(metric)量化.正确地检测和量化 Code Smell 可以指导软件重构,提高软件的可用性和可靠性.

然而,现有 Code Smell 度量不足以理解 JS 程序的复杂性和程序质量,常见的 Code Smell 度量很少被 JS 相关的研究使用[4],需对其进行开发或改进,以适应 JS.目前已出现面向 JS 的 Code Smell 检测工具,例如 SonarQube, JSNose[1],它们仅面向微观层面上(即函数、语句)的内聚或耦合问题.

JSNose 主要讨论了函数、代码块及对象内部的 Code Smell,并引入了 3 种通用类 Code Smell 的检测方式来检测对象内部的问题,例如过大对象和不当的继承,但研究不涉及多个对象间的设计问题.研究还提及了一种文件层面的 Code Smell,它面向 JS,HTML 和 CSS 这 3 种语言在代码中混写的耦合问题.Saboury 等人[5]根据语言标准的新变化,扩充并修正了 JSNose 的定义,此类修正主要发生在语法和语句层面上,涉及作用域和闭包等问题.该研究还就 Code Smell 对 JS 代码易错性(fault-proneness)的影响排序,结果显示,部分涉及不当赋值的 Code Smell 更易引发程序错误.

上述微观研究可以提出程序实现的重构建议、局部提升程序的质量,但无法有效分析粒度更大的(例如类、模块)设计问题.Palomba 等人[6]发现:因为大粒度的代码组件过于复杂,模块和类层面的 Code Smell 相比函数 Code Smell 更难被有效地识别,解决这类问题的效率和成功率极低,因此需要理论和工具辅助.

类(class)是 JS 的一种常见设计模式,在 Silva 等人[7]抽选的 60 个 JS 程序样本中,68%使用了类,34%较频繁或频繁地使用了类.优秀的类设计体现为高内聚和低耦合[8],内聚(cohesion)和耦合(coupling)可用于衡量面向对象程序功能分布的合理性,内聚是软件组件内功能的关联程度,耦合是软件组件间交互的频繁程度.

由于 JS 类的信息检测存在困难,导致部分 Code Smell 的通用检测方法难以复用,因此 Code Smell 的相关工作仍未充分涉及 JS 类的常见设计问题.提出一个 JS 类的内聚耦合 Code Smell 检测方式是迫在眉睫的任务.

本文的主要贡献为:

(1) 结合代码的文本和结构信息,提出了一个检测 JS 类的内聚耦合 Code Smell 的静态分析方法 JS4C,它可以量化 Blob、Feature Envy(简称 FE)和 Dispersed Coupling(简称 DC)这 3 种 Code Smell,并检测 JS 开源软件中的内聚耦合问题.

(2) 细化并改进了类耦合 Code Smell 的检测方式:首先,根据耦合、内聚的设计问题对 Fowler 定义的 22 种 Code Smell 进行了分类,对比并区分了 FE 的两类检测方式;其次,引入了 DC 细化耦合问题检测的粒度,改进了其 CDISP 度量,以适应 JS 的弱类型语言特性.

(3) 指出了文本信息源对 JS Code Smell 检测的重要性.实验发现,引入文本特征可以提升 Code Smell 的检测效果.

本文第 1 节介绍本文所涉研究领域的相关工作以及其中存在的困难和问题.第 2 节介绍本文所涉 3 种 Code Smell,简述检测的各个阶段,包括预处理、检测和结果输出,是对检测流程的概览.第 3 节研究 3 种 Code Smell 的文本和结构方式检测算法,并给出一个综合判定策略.第 4 节给出实验,以验证检测结果的准确性,并探究对检测准确性造成负面影响的因素.最后得出结论,并讨论今后的工作.

1 相关工作

1.1 检测JS类及相关信息

图 1 为 3 个拥有相同属性和方法的 Rental 类,图 1(a)的程序是一个 Java 类,其余两段程序为 JS 类,JS 类的

实现参考了 Fowler 的影音店案例[9,10]和 JS 类检测的文献[7].

```
1.    public class Rental {
2.      private Movie movie;
3.      private int daysRented;
4.      public Rental(Movie movie,int daysRented){
5.        this.movie=movie;
6.        this.daysRented=daysRented;
7.      }
8.      public int getDaysRented(·){return daysRented;}
9.      public Movie getMovie(·){return movie;}
10.   }
```

(a) 一个 Java 类

```
1.    function Rental(data){
2.      this._data=data;
3.    }
4.    Rental.prototype.getMovie=function(·){
5.      return this._data_.movie;
6.    }
7.    Rental.prototype.getDaysRented=function(·){
8.      return this._data_.daysRented;
9.    }
```

(b) ES2015 标准发布前的一种 JS 类设计模式

```
1.    class Rental{
2.      constructor(data){this._data=data;}
3.      get daysRented(·) {
4.        return this._data_daysRented;
5.      }
6.      get movie(·){
7.        return this._data.movie;
8.      }
9.    }
```

(c) 一个利用糖衣语法实现的 JS 类

Fig.1 Demonstration code showing Java and JS class implementations

图 1 Java 和 JS 类的实现代码示例

JS 类是基于 JS 原型继承(prototype-based inheritance)构建的一种经典模型[11],它的实现可分为两类[12].

(1) 在早于 ES2015 的语言标准中,类是基于函数的设计模式,实现方式不唯一,图 1(b)是一种常见的实现.

(2) 如图 1(c),ES2015 及更新的标准用糖衣语法(syntactic sugar)简化了类的实现方式.

检测 JS 类首先需要定位构造器(constructor,例如图 1 中的灰色高亮部分)和成员属性(例如*_data*),再遍历函数的原型链(例如图 1(b)的 *Rental.prototype*)搜索其成员函数和子类.构造器是类的显著特征,可用于创建对象.定位构造器,可通过查找类的实现函数,或查找构造器的调用语句(例如 new 语句).目前,已有效果较好的、基于抽象语法树遍历的开源 JS 类检测工具 JSDeodorant[13]、JSClassFinder[7].

本文需要检测的 JS 类信息为类的成员(即函数和属性)、类的函数及其签名、类引用的外部数据提供者(foreign data provider,简称 FDP)和类引用的内部成员.某类的“外部数据”指的是软件系统中非本类的类成员.分析引用关系,可通过遍历代码中的访问操作(access)实现.访问操作是指对对象的引用、对其成员属性的引用(reference)或对其成员函数的调用(call).

检测 JS 类及相关信息的挑战包括:

(1) 检测难以适用于所有的类实现方式.实现类的设计模式,有至少 5 种较为常用的、适用于不同版本语言标准的方式[7,13],给软件组件的判定和标准化带来困难.

(2) 检测类与类间的关系困难.受 JS 的语言特性限制,类成员属性的类型在静态分析过程中是未知的.类型信息的缺失,导致分析引用关系的困难.这种缺失无法使用动态分析完全弥补,因其依赖爬虫,不适用于运行在服务端的 JS 程序.

1.2 结构分析

结构分析是最常见的 Code Smell 检测方法,通常包括 2 个阶段,即信息收集和检测判定:在信息收集阶段,分析工具通过遍历抽象语法树获取检测对象的结构信息;在检测判定阶段,需要根据度量的定义将收集得到的信息计算为度量值,再依据规则判定是否存在 Code Smell.规则根据 Code Smell 的定义设计,它通常由一系列度

量阈值条件构成,规则也可以包含其他非数值型特征(例如函数名等文本特征)[14].

本文检测 3 种 Code Smell,包括 Blob,FE 和 DC.Blob 体现低内聚,其余二者体现高耦合,定义分别为:

(1) Blob 表现为一个类,它实现了系统中的多种不同职责、体积庞大且复杂,其函数间的内聚性低[6,14,15].

(2) DC 表现为一个函数,与其所在类相比,它更依赖多个非所在类的操作[16,17].

(3) FE 表现为一个函数,与其所在类相比,它更依赖某个特定的、非所在类的操作[6,15].

根据软件组件的粒度及设计问题类型,表 1 给出了 Fowler[9]定义的 22 种 Code Smell 的分类情况.其中,FE 因同时涉及方法和类的耦合,故出现两次.God Class 亦称 Blob,后文均使用 Blob 的表述.

Table 1 Code Smell classification by the types of design problems and their granularity

表 1 以设计问题种类及其粒度为依据的 Code Smell 的分类

<table>
<tr><th>粒度/设计问题</th><th colspan="2">低内聚</th><th colspan="2">高耦合</th><th>其他</th></tr>
<tr><td>方法/函数</td><td colspan="2">Data Clumps,
Long Method,
Long Parameter List</td><td colspan="2">Feature Envy</td><td>Switch Statements</td></tr>
<tr><td rowspan="2">类</td><td>与类的继承无关</td><td>与类的继承相关</td><td>与类的继承无关</td><td>与类的继承相关</td><td rowspan="2">Alternative Classes with Different Interfaces,
Data Class,
Primitive Obsession,
Temporary Field</td></tr>
<tr><td>God Class</td><td>Refused Bequest</td><td>Feature Envy,
Middle Man,
Inappropriate Intimacy,
Divergent Change,
Message Chains</td><td>Shotgun Surgery,
Parallel Inheritance Hierarchies</td></tr>
<tr><td>包/软件系统</td><td colspan="2">–</td><td colspan="2">–</td><td>Comments,
Duplicate Code,
Dead Code,
Speculative Generality,
Incomplete Library Class</td></tr>
</table>

Silva 等人[7]指出:JS 软件项目极少使用类的继承特性,在样本中仅占 8%.故本文仅讨论与类继承无关的 Code Smell.Palomba 等人[18]提出,Refused Bequest 同 Message Chains 以及 FE 同 Inappropriate Intimacy 具有极高的同现性.故本文不涉及 Message Chains 和 Inappropriate Intimacy.林涛等人[19]指出:Divergent Change 与 Shotgun Surgery 矛盾对立、存在权衡关系,且其检测过程需要对多版本进行分析.故不列入研究范围.Middle Man 是一种耦合 Code Smell,表现为类代理(delegate)了过多其他类的操作,可视作 FE 的一个特殊情况.

综上,在表 1 的 22 种 Code Smell 中,本文聚焦 Blob 与 FE.因 FE 的量化标准不一,故需分类讨论.根据表 2,标准可分为两类:其一,确定被检类与一或多个类发生耦合,但不确定与其耦合的类;其二,确定被检类与一个类发生耦合,并确定与其耦合的类.为了细化耦合问题的检测,本文对 FE 取后者的定义及检测方式.为了覆盖同多个类发生耦合情况,本文引入 DC[16].

Table 2 Comparison of three Code Smells related to class coupling

表 2 3 种类耦合相关的 Code Smell 对比

检测方式/检测细节	FE(Lanza 等人[14])	FE(Fokaefs 等人[20])	DC(Palomba 等人[16])
耦合对象	一或多个类	一个类	多个类
确认耦合对象类名	否	是	否
度量参数	3 个:外部数据访问次数(ATFD)、本地属性访问比例(LAA)、外部数据提供类个数(FDP).	2 个:对特定非所在类的属性及方法访问的最高次数、对所在类本地属性及方法的访问次数.	2 个:耦合强度,即外部数据及函数访问次数(CINT)、耦合分散度,即外部数据提供类个数与 CINT 的比值(CDISP).
本文是否采用	否	是	是

结构分析的挑战,是由 JS 语言特性和 Code Smell 度量指标的特性共同构成的,它们包括:

(1) JS 的语言特性导致结构分析所需的精确信息难以获取,检测耦合 Code Smell 需要推断类型或依赖其他信息源.

(2) 部分 Code Smell 的量化标准不一.

(3) 类的 Code Smell 检测方式均针对以 Java 为代表的非解释型语言,对于 JS,需要改进部分度量.

1.3 文本分析

文本分析与结构分析的主要区别在于信息收集阶段,前者不依赖或较少依赖抽象语法树中的结构信息,而是将源码视作文本片段,并使用文本挖掘算法计算文本相似度等度量信息.

Palomba 等人[6,15,16]运用结构分析和文本分析,对 Java 软件项目的多个历史版本进行了挖掘,研究分析了两种检测方式获取的 Code Smell 强度,得出了其解决的难易程度、变化的趋向、产生的原因等性质.研究指出,基于结构方式和文本方式的检测算法互补,文本方式是重要的信息源,将两种信息源结合的工作较少.

因为结构分析方法在检测 JS 变量的类型信息时,所能获取的信息远不如 Java 项目详实,所以文本分析和结构分析的互补对 JS Code Smell 的检测更为重要.

文本分析方法可分为有监督和无监督两类.有监督的文本分析需要额外的人工干预来预先标注文本,以形成带标签(Label)数据集,故不适用于 Code Smell 的检测.在无监督的文本分析方面,Blei 等人[21]提出了潜在语义分析(LDA)算法,该算法被应用于 Bavota 等人[22]检测 Code Smell 的方法中;Le 等人[23]基于 Google 开源的、运用深度模型的 Word2Vec 词向量算法,提出了将文档用向量表示的算法.

LDA 是一种常用的文档主题生成式模型,包含词、主题和文档这 3 层结构.LDA 使用词袋模型表示一个文档,并将一个文档视作隐含主题的有限混合,其中的每个单词由一个主题生成,文档之间的关联可以由主题间的关联决定.LDA 具备比 LSA 等基础主题建模方法更强的拟合能力.

Word2Vec 算法可利用神经网络语言模型训练过程中获得的权重矩阵参数,将词语转化为向量.Skip-gram 是 Word2Vec 算法使用的一种语言模型,它根据当前词汇预测上下文.Skip-gram 模型共有 3 层,即输入层、隐藏层(hidden layer)、输出层.神经网络通过调整隐藏层至输入层、输出层的权重矩阵进行训练,调整训练的过程可概括为:假如若干词具有近似的输出,则可反推词间具有较高的相似性.基于 Skip-gram 模型,Mikolov 进一步提出了 Doc2Vec,它基于词向量表示文档向量.本文采用 PV-DBOW(distributed bag of words version of paragraph vector)模型,其设计思想和 Skip-gram 相近,即:通过预测文档的内容训练一个矩阵,此矩阵即为文档矩阵.通过计算文档矩阵间的余弦相似度,可获取相似度特征.

2 检测流程

检测流程包括 4 个阶段:信息抽取阶段、预处理阶段、检测阶段、判定阶段,如图 2 所示.

(1) 信息抽取阶段:从软件项目的开源社区获取特定 Release 版本的源码,配置类检测引擎的运行参数后开始类的检测.类检测引擎基于 JSDeodorant[13]改进,JSDeodorant 是开源 JS 类检测工具,文献报告其平均表现可达到 95%的精确率和 98%的召回率.在此基础上,本文借助 Google Closure Compiler[24]分析源码所得的 JS 语句类型信息,通过扩展对象类型推断[25],实现对类信息的检测.

(2) 预处理阶段:本阶段为检测阶段所需的输入做准备.对于结构分析,需对检测到的 JS 类信息去重和计算频次,以便用于度量指标;对于文本分析,需将文本标准化.文本标准化的具体流程如下:

- 分词.对驼峰和下划线命名方式的变量名进行分词,将所有英语字符都转换为小写形式.
- 去除停用词.将与业务逻辑无关的保留关键字、英语停用词、特殊字符等内容剔除.
- 提取词干.去除单词的词缀,得到其词根.

(3) 检测阶段:本阶段对输入数据运行检测算法,并计算动态阈值,最终输出值域为[0,1]的强度值.其中,本文对 FE 运用了 Fokaefs 等人[20]提出的检测算法,对 DC 运用了经改进的非严格 NSCDISP 度量,对 Blob 运用了 LCOM5 度量[26].在基于相似度的文本检测方面,本文对 FE 和 Blob 的检测使用了 Doc2Vec 算法[23],并以 LDA 算法的结果作为对照组.由于文本分析对 DC 的检测效果不佳,故未予采用.

(4) 判定阶段:对于一个代码片段,检测阶段可输出多个强度值.根据 Code Smell 之间的关联和结构和分析方法的特点、优劣,本文提出了一种综合判定策略,最终得出该代码片段的 Code Smell 的种类及值域为[0,1]的强度值,强度值越高,Code Smell 越严重.

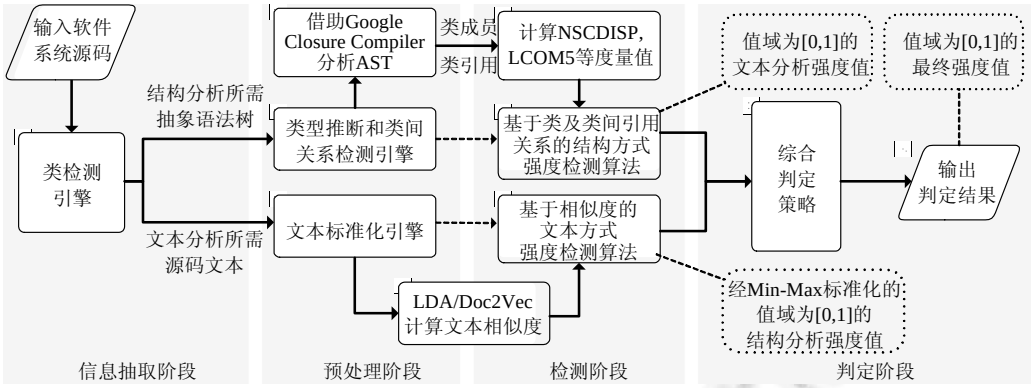

Fig.2 Code Smell detection process of JS4C

图 2 JS4C 的 Code Smell 检测流程

3 检测算法和判定策略

3.1 基于结构分析的强度检测算法

Code Smell 检测的传统方法使用代码的结构信息作为信息源[6],图 3 展示了本节涉及的结构信息和度量.

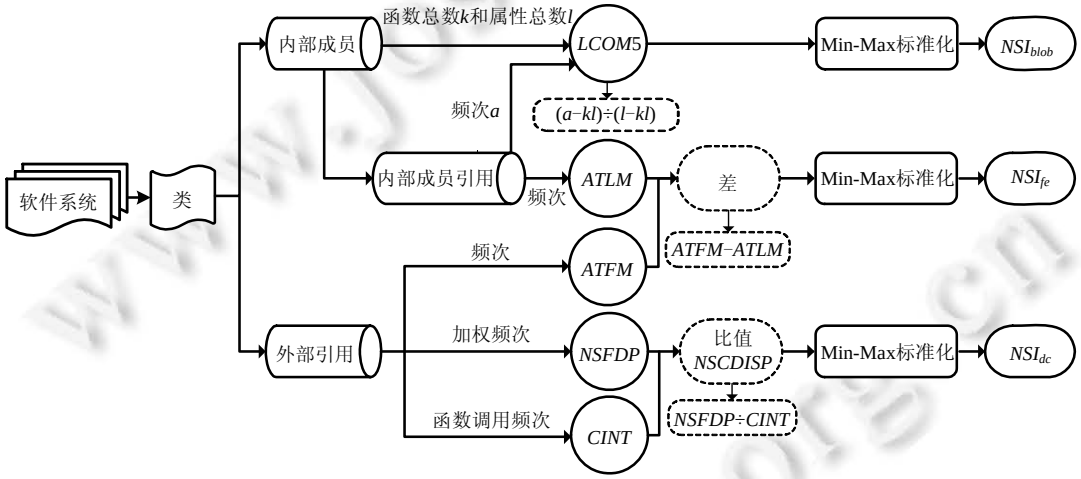

Fig.3 Overview of structural analysis

图 3 结构分析概览

3.1.1 结构方式检测 DC

Palomba 等人[16]对 Java 的 DC 检测使用了 CINT(coupling intensity,耦合强度)、CDISP(coupling dispersion,耦合分散度)和 FDP(foreign data provider,外部数据提供类)这 3 种度量.其中,CINT 为被检测函数中经去重的函数调用次数,FDP 指标用是所有被访问的数据所属类的去重集合.CDISP 的计算公式如式(1)所示.

$$CDISP(f)=\frac{N(FDP(f))}{CINT(f)} \tag{1}$$

由于 JS 的语言特性限制了类型检测,即便类型推断可确定部分类型,也无法确定全部函数的入参类型和数据访问操作的变量类型.CINT 和 CDISP 的计算都依赖类型判定,若不作改进,检测难以进行.

据此,本文提出非严格的 NSCDISP(non-strict coupling dispersion),它的非严格是针对 FDP 和 CINT 必须精确检测函数入参(input parameter)的类型而言的.尽管引用的对象类型难以精确检出,但类中的数据访问操作总

数和类的成员尚可明确,仍可利用以上信息得知引用操作的数量.

CDISP 的计算依赖 FDP 和 CINT.本文对 CINT 的检测不作改动,使用非严格的 FDP 个数检测方式:对于不涉及函数入参的数据访问,沿用原 FDP 的检测方式,若无法检测到所属类,则将其视作一个单独的 FDP;对于涉及函数入参的数据访问,则计算 WRFDP.

WRFDP(*FDP* with weighted reference)是面向入参外部引用权重的度量,具体方式为:对于函数 f,有入参集合 P_f.计算 f 在所属类中被调用的总次数 ntc(number of total calls).遍历 P_f,对 P_f 的每个元素 p_{f_i},计算 p_{f_i} 被调用时入参形参的所属类为外部类的次数 ncfp(number of calls as foreign param).计算 ncfp 与 ntc 比值,其值域在[0,1]之间.ncfp 计算的基本思路是:分析被检函数的每次调用,若调用的入参不源于所属类,则计入 ncfp.

对于被检函数的每个入参,均计算其 WRFDP,如式(2)所示.

$$WRFDP(f,p_{f_i})=\frac{ncfp(p_{f_i})}{ntc(f)} \tag{2}$$

非严格的 NSFDP 的计算公式如式(3)所示.

$$NSFDP(f)=N(FDP(f))-N(P_f)+\sum WRFDP(f,p_{f_i}) \tag{3}$$

本文将非严格的 NSCDISP 作为 DC 的强度(intensity)值 I_{dc},其计算公式为

$$I_{dc}(f)=NSCDISP(f)=\begin{cases}0, & CINT(f)<\text{动态阈值}\\ \dfrac{NSFDP(f)}{CINT(f)}, & CINT(f)\geqslant\text{动态阈值}\end{cases} \tag{4}$$

NSCDISP 和 CINT 的检测阈值(文献[16]中的 HIGH)取系统检测结果的平均数(AVG)与标准差(STDEV)之和[17],未达到阈值的相关度量值将被改为 0,下同.

在计算出软件系统内的全部 I_{dc} 后,对于该系统的全部函数$\{f_1,f_2,f_3,\ldots,f_i\}$,可以得到一个集合 $SI_{dc}=\{I_{dc}(f_1),I_{dc}(f_2),I_{dc}(f_3),\ldots,I_{dc}(f_i)\}$,对所得结果进行 Min-Max 标准化,使其值域落入[0,1]区间,得出最终的强度值集合 NSI_{dc} (normalized structural intensity),如式(5)所示.

$$NSI_{dc}(f_i)=\frac{I_{dc}(f_i)-\min(I)}{\max(I)-\min(I)} \tag{5}$$

3.1.2 结构方式检测 FE

对于被检类 $C_{current}$,遍历系统的类全集 C.对于 C 中的每个元素 C_i,检测 $C_{current}$ 对 C_i 成员(函数、属性)经去重后的访问次数 a_i,并依据 a_i 的值对 C 中的类从大到小排序,若排序第一的类 C_{top} 不等价于 $C_{current}$,则判定存在 FE.将 $C_{current}$ 对 C_{top} 的成员访问次数 a_{top} 记为 ATFM(access to foreign members),将 $C_{current}$ 对自身的成员访问次数 $a_{current}$ 记为 ATLM(access to local members),FE 的强度为式(6)[20].

$$I_{fe}=ATFM(C)-ATLM(C) \tag{6}$$

若 $I_{fe}>0$,即可判定 FE 存在.用类似第 3.1.1 节的方式对所得结果进行 Min-Max 标准化,得到结果集合 NSI_{fe}.

3.1.3 结构方式检测 Blob

利用结构方式检测 Blob 有两个角度:类的体积和类的内聚性[26].前者根据代码长度或类成员的总数,后者采用 LCOM5(lack cohesion of method 5)度量.由于本文的主题仅和内聚、耦合相关,且类体积相关的度量需指定固定的阈值,故不将类的体积纳入检测考虑的因素.

对于类 C,获取其函数成员总数 k、属性成员总数 l、被访问的自身成员(函数、属性)经去重后的总数 a,定义 LCOM5 为式(7),其值域在[0,2]间,检测阈值取第三分位点(75%)的值[27].

$$NSI_{blob}(C)=LCOM5(C)=\frac{a-kl}{l-kl} \tag{7}$$

用类似第 3.1.1 节的方式对所得结果进行 Min-Max 标准化,得到结果集合 NSI_{blob}.

3.2 基于文本分析的强度检测算法

图 4 展示了文本分析的方式,以经文本标准化引擎处理的全部程序文本为训练集,将每一个类作为一篇文

档,训练 Doc2Vec 模型.通过该模型,可计算任意两组代码段文本的余弦相似度.余弦相似度的值域落在[−1,1]间,负值的物理意义即为负相关.由于负值偶发,且本文不关注代码段负相关的信息,参照 Positive PMI[28,29]对类似问题的处理方式,将余弦相似度的负值视为 0,即对于文本段 T_a,T_b,其中 a 与 b 为任意代码片段,其相似度定义如式(8)所示.

$$PSIM(T_a,T_b)=\begin{cases}0, & SIM(T_a,T_b)<0\\ SIM(T_a,T_b), & SIM(T_a,T_b)\geqslant 0\end{cases} \tag{8}$$

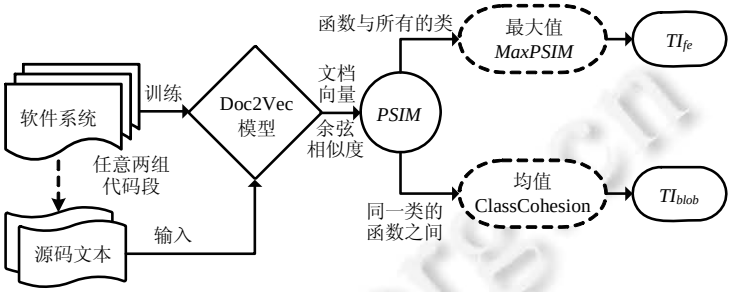

Fig.4 Overview of textual analysis

图 4 文本分析概览

本文采用类似的方式,利用 LDA 计算文本的相似度,并将其作为对照组,相似度值域为[0,1].

本文参考 Palomba 等人[18]使用系统内全部非零度量值的中位数(non-null median)作为 TI_{fe} 和 TI_{blob} 的阈值.

3.2.1 文本方式检测 FE

此方法的基本思路是:对于被检函数 f,若存在一个非所属类,它与 f 的相似度比 f 同所属类的相似度相比更高,则存在耦合,可判定检出 FE.定义 f 所属的类为 C_O,计算 f 所有类的集合 SC_{ALL} 中每个类的文本相似度.即:对于任意 $C_i\in SC_{ALL}$,计算 $PSIM(f,C_i)$,记录最大值为 $MaxPSIM$,如式(9)所示.

$$MaxPSIM(f,SC_{ALL})=\mathrm{MAX}(PSIM(T_f,T_{C_i})),C_i\in SC_{ALL} \tag{9}$$

对于 FE,定义 Code Smell 的强度[6]为式(10).

$$TI_{fe}(f,SC_{ALL},C_O)=MaxPSIM(f,SC_{ALL})-PSIM(T_f,T_{C_O}) \tag{10}$$

3.2.2 结构方式检测 Blob

此方法的基本思路是:类的函数间越不相关,则类的内聚性越低,Blob 的强度越大.

对于被检类 C 及其函数成员的集合 F,如式(11)所示,定义 Code Smell 的强度[6]为

$$TI_{blob}(C)=1-ClassCohesion(C) \tag{11}$$

其中,ClassCohesion 为类函数间的相关性计算函数,其定义[15]如式(12)所示.

$$ClassCohesion(C)=\underset{i\neq j}{mean}\,PSIM(f_i,f_j),f_i,f_j\in F \tag{12}$$

3.3 综合判定策略

本文按设计问题将 Code Smell 分为内聚类和耦合类,图 5 中的 Blob 属于内聚类,图 6 中的 DC 和 FE 属于耦合类.

判定分为两个步骤:首先,筛选 Code Smell 度量强度超过阈值的检测对象;其次,根据文本和结构方式的强度确定 Code Smell 的最终强度.对于 Blob,通过结构和文本分析检测到的 Code Smell 之并集识别.如式(13)所示,强度值 I_{blob} 取两者的最大值:

$$I_{blob}=\mathrm{MAX}(TI_{blob},NSI_{blob}) \tag{13}$$

对于 DC,本文采用结构分析识别,即 NSI_{dc}>动态阈值,如图 6.强度值 I_{dc} 取结构分析的强度如式(14)所示.

$$I_{dc}=NSI_{dc} \tag{14}$$

对于 FE,本文将存在结构耦合(0<*CINT*)作为检测 FE 的前置条件,并优先取文本方法的检测值.如式(15)所示,强度值 I_{fe} 为

$$I_{fe}=\begin{cases}TI_{fe}, & 0<CINT\text{且}TI_{fe}>0\\ NSI_{fe}, & 0<CINT\text{且}TI_{fe}=0\\ 0, & 0\geqslant CINT\end{cases} \tag{15}$$

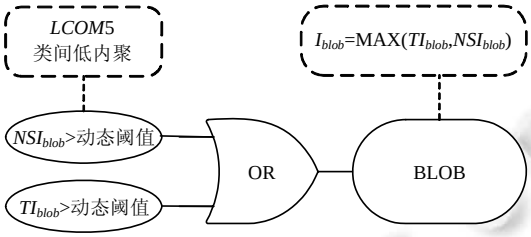

Fig.5 Unified identification strategy of cohesion design problem

图 5 内聚设计问题的综合判定策略

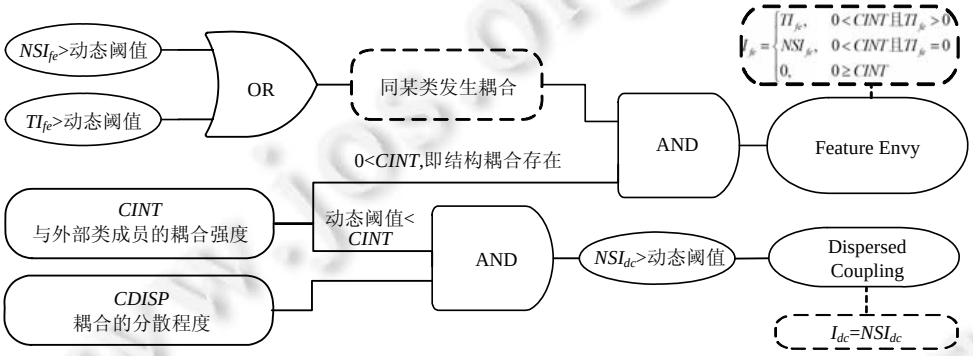

Fig.6 Unified identification strategy of coupling design problem

图 6 耦合设计问题的综合判定策略

3.4 可行性分析

本文的先进性体现在以下几处:在结构分析方面,本文讨论并选定了 3 种 Code Smell 的检测方式和阈值,针对 JS 检测任务提升了 DC 检测涉及的 FDP 和 CDISP 指标的宽容度,使其适应类型不明的情况;在文本分析方面,本文使用了更优的文本分析算法,并利于文本语义特征实现文本和结构分析的检测结果互补.针对两种分析方法的特点,本文制定了 3 种 Code Smell 的综合判定策略.

将文本分析和结构分析结合是可行且有效的.

- 一方面,代码组件的文本中蕴含业务逻辑和组件功能(例如 service,manager)信息,这些信息在结构分析时往往难以被充分利用[30].已有学者尝试改善这一情况,例如:Moha 等人[26]检测 Blob 时,在结构方式的基础上根据类名判断组件功能;Palomba 等人[15]通过实验得出文本和结构分析的检测结果互补,相互结合可以获得更好的检测效果,将两种信息源结合的工作较少.
- 另一方面,JavaScript 的类型系统和 Java 等强类型语言不同,在变量声明时无需明确类型,导致传统的静态分析难以奏效,无法获得像强类型语言一样详实的类型信息.

判定 Blob 使用文本和结构方式的最大值的原因是:(1) Blob 的结构方式只需要检测访问本地和外部数据的次数,对于外部数据的访问,无需明确操作涉及的类,因此获得的结构信息是完整的;(2) 本文的强度值均标准化至[0,1]区间,可以直接比对.

判定 FE 时,之所以优先取文本方法的检测值,是因为结构方式检测 FE 需要确定耦合对象所属的类.由于 JS

使用弱类型系统,变量的类型在分析中仅靠推断得出,因此这一结构信息可能检测不全.文本方式根据语义相似度确定是否耦合,不受此限制,因而更可靠.

判定 FE 时,使用 CINT 作为阈值的原因为:一方面,计算 CINT 无需确定耦合对象所属的类,因此该度量的值是可靠的;另一方面,存在业务逻辑类似但实则毫无关联的代码,它们可能会导致文本方式的误判,例如将代码拷贝误判为耦合,使用 CINT 阈值可以应对这种情况.

4 实 验

为了验证 JS4C 的有效性,本文以表 3 中的开源项目作为测试数据集,对 JS4C 的检测结果进行评估对比.实验主要寻求以下几个问题的解答.

Q1:JS4C 能否准确检测内聚和耦合的设计问题?

Q2:JS4C 有哪些已知问题?是否仍有改进空间?

Q3:文本检测算法是否提升了 JS4C 的检测效果?

Q4:哪些因素会影响实验评估结论的有效性?

Q5:如何基于 JS4C 给出 JavaScript 类的重构建议,并实现其应用意义?

Table 3 Open source project dataset used for experiment

表 3 实验所用开源项目数据集

项目名	版本	类总数	类的函数总数	功能描述	运行环境
Awesome-qr.js	1.2.0	9	61	二维码生成	浏览器、服务器端
Three.js	r95	232	799	三维引擎	浏览器端
PDF.js	1.1.1	535	1 574	PDF 文档阅读	浏览器端
FloraJS	3.1.1	24	383	物理引擎	浏览器端
Pixi.JS	3.0.2	88	610	HTML5 2D 绘图	浏览器端
Brackets	1.3	160	4 740	集成开发环境	桌面系统

4.1 实验过程

为了保证基础数据的准确性,实验首先对基础度量数据进行了验证.由于缺乏对比的工具,本节实验参照同类 Code Smell 的工作[1,3],将对比基准确定为人工检测.检测工作由本文的第一作者及一名拥有 3 年工作经验的 Web 应用开发者分别独立进行后,针对分歧讨论完成.

其中,Doc2Vec 和 LDA 的相似度值分别使用 Deeplearning4j 和 JGibbLDA 库计算,它们被软件分析相关研究[31]和开源社区广泛验证和使用,因此文本分析的人工验证仅涉及输入数据和运算的准确性.结构方式的度量值均源于抽象语法树的结构信息,验证工作将人工计算的结果与工具的结果进行了对比.

获取可靠的度量值后,实验需要进一步验证 JS4C 对实际内聚、耦合设计问题的检测效果.本文对数据集中的项目采用了与度量值验证相同的人工检测流程,特别地,对于耦合问题,除了判定有无设计问题外,还需识别出所有人工判定发生耦合的对象,否则视为错判.

为了回答 Q1 和 Q2,本文使用精确率(precision)和召回率(recall)度量检测的效果,指标的数值越大,表明检测的效果越好.将 Code Smell 检测视作一个二分类问题,将存在 Code Smell 的类归为正样本类,不存在 Code Smell 的类归为负样本类.令预测为正的正样本数量为 TP(true positive,正确率)、预测为正的负样本数量为 FP(false positive,误报率)、预测为负的正样本数量为 FN(false negative,漏报率),精确率和召回率分别如式(16)、式(17)所示:

$$Precision = \frac{TP}{TP + FP} \tag{16}$$

$$Recall = \frac{TP}{TP + FN} \tag{17}$$

本文对数据集中全部的项目进行了检测,本节以 Awesome-qr.js 案例和 Three.js 为例,说明实验过程.本节的

度量数据在第 4.2 节中的表 7~表 9 展示.

Awesome-qr.js 的 1.2.0 版本[32]约有 1 500 行代码,可在客户端或服务端运行.它拥有两个工具类、7 个功能类,且多个类具有低内聚特征.

表 4 列出了对该项目受 FE 和 DC 影响的全部函数及检测结果.由于上述函数的代码及变量命名均无显著的业务特征(例如 mod 和 multiply)或特征过多、分散(例如 draw),甚至未出现在软件项目中,故文本分析难以确定目标的耦合类.然而,它们的确存在耦合问题,对于这类较为困难的任务,JS4C 体现出了良好的适应性.

Table 4 Detection results and effectiveness of FE and DC in Awesome-qr.js

表 4 Awesome-qr.js 中 FE 和 DC 的检测结果和检测效果

类名	函数名	I_{dc}/耦合对象	NSI_{fe}/耦合类	I_{fe}/耦合类(Doc2Vec)	I_{fe}/耦合类(LDA)	人工判定
QRPolynomial	mod	0.4/QRMath,入参 *e*	0/无	0/无	0/无	QRMath
QRPolynomial	multiply	0.5/QRMath,入参 *e*	0/无	0/无	0/无	QRMath
Drawing	draw	0/无	0/无	0/无	0/无	QRCodeModel
QRCodeModel	make	0.33/QRRSBlock, QRUtil	0.17/QRRSBlock	0.17/QRRSBlock	0/无	QRRSBlock
AwesomeQRCode	makeCode	0.4/Drawing, QRCodeModel	1/Drawing, QRCodeModel	1/Drawing, QRCodeModel	1/Drawing, QRCodeModel	QRCodeModel
QRUtil	getLostPoint	0/无	0/无	0/无	0	QRCodeModel

表 5 列出了该项目 Blob 的检测结果.在上述检测结果中,对于 QR8bitByte,文本分析未检测到 Code Smell,而结构分析检出了最高强度的 Code Smell.因为类中定义了多个未经函数成员访问的属性,内聚性不足,但函数成员的名称却与业务逻辑有关.对于 AwesomeQRCode,它未包含任何属性成员,故结构分析会失效,但文本分析却可以检测出它实现了不相关的多种职责.

Table 5 Detection results and effectiveness of Blob in Awesome-qr.js

表 5 Awesome-qr.js 中 Blob 的检测结果和检测效果

类名	代码行数	NSI_{blob}	TI_{blob}(Doc2Vec)	I_{blob}(Doc2Vec)	TI_{blob}(LDA)	I_{blob}(LDA)	人工判定
QRCodeModel	533	0.58	0.51	0.58	0.61	0.61	有
Drawing	327	0.68	0.74	0.74	0.61	0.68	有
QRPolynomial	44	0	0.29	0.29	0	0	有
QRBitBuffer	28	0.26	0.21	0.26	0	0.26	有
QR8bitByte	41	1	0	1	0	1	有
AwesomeQRCode	55	0	0.46	0.46	0.56	0.56	有
QRRSBlock	197	0	0	0	0	0	无
QRUtil	213	0	0	0	0	0	有
QRMath	18	0	0	0	0	0	无

Awesome-qr.js 的体积较小,在相关性分析中难以获取足够的有效数据,需要使用规模较大的软件演示后续步骤.Three.js[33]是利用 WebGL 和 HTML5 特性实现的开源浏览器端三维引擎,截至 2018 年 9 月,该项目已有近 25 000 次代码提交(commit)、88 个发布(release)版本和近 1 000 名代码贡献者(contributor).本文利用 JS4C 对 Three.js r95 版本代码中的 232 个类进行了检测,并分析 Code Smell 的分布规律及它们之间的关系,其中,共 103 个类涉及本文讨论的 3 种 Code Smell.

对于系统内所有的类,获取 3 种 Code Smell(Blob,FE,DC)强度的集合 BLOB,FE,DC.本文利用 Spearman 等级相关方法(Spearman's rank correlation coefficient)[34]分析三者间的两两相关性,该方法可用于未知概率分布的两组顺序数据,取上述两组数据作为输入值,可输出其相关系数ρ及相关性的显著程度 P.判定显著程度前,需要选定一个显著等级(significant level)值α,实践中,α通常取值 0.05[6].在该取值下,当 $P<\alpha$时,可认为两组数据的差异具有显著性,即具有统计意义;当 $P<0.01$ 时,可认为两组数据的差异非常显著.相关系数ρ利用单调方程评价两个统计变量的相关性,它的值域为[–1,1].如表 6 所示,ρ值可对应相关性,当数据中没有重复值,且两组变量完全单调相关时,ρ相关系数则为+1 或–1.

Table 6 Spearman's rank coefficient value ρ and its correlation level

表 6 Spearman 秩相关系数ρ的值及其相关性

ρ值范围	相关性
[0.8,1.0]	极强相关
[0.6,0.8)	强相关
[0.4,0.6)	中等强度相关
[0.2,0.4)	弱相关
[0.0,0.2)	极弱相关

本文对 Three.js 的 3 种 Code Smell 检测结果进行两两分析,得出结论:FE 和 DC 呈现弱强度的负相关性,其中ρ为−0.30(P=0.001);BLOB 和 FE 呈现中等强度的正相关性;BLOB 和 DC 的差异不具统计意义.

本文进一步探究耦合设计问题和内聚设计问题的关系.

将 Code Smell 分为两类,令内聚 Code Smell(Blob)为一类,令耦合 Code Smell(FE,DC)为另一类.类低内聚的强度 I_{LC} 取类的 I_{blob}.一个类具有多个函数,因而具有多个 FE 与 DC 强度值,对于耦合 Code Smell,本文取其均值 $mean(I_{fe})$,$mean(I_{dc})$,类高耦合 I_{HC} 的强度优先考虑同一个类发生耦合,即取 $mean(I_{fe})$,若 $mean(I_{fe})\leqslant 0$,则取 $mean(I_{dc})$.对于系统内所有的类,可以获得低内聚强度的集合 LC 和高耦合强度的集合 HC.

本文利用 Spearman 秩相关系数分析 LC 和 HC 的相关性,仅考虑 I_{HC},I_{LC} 均大于 0 的情况,得到 P 值远小于 0.05(6.3e−18),ρ为 0.46.实验发现,该项目耦合和内聚的强度具有弱正相关性.

将低内聚、高耦合类交集的元素个数与低内聚、高耦合类并集的元素个数的比值作为同现率,得出同现率为 66.88%.本文进一步使用 Wilcoxon 秩和检验(Wilcoxon rank-sum test)[35]分析两组数据的总体分布是否差异显著:若差异不显著,则相关性和同现率无实际意义.秩和检验亦输出显著程度 P,显著等级取值和α相同.分析得出,LC 和 HC 的分布显著不同(P=6.27e−34).

4.2 实验结果

通过与第 4.1 节类似的方式,本文对数据集中的所有项目进行了检测,检测结果在表 7~表 9 中列出.在表 7 和表 8 中,为了验证文本方式对结果的贡献,实验单独列出了去除文本方式后的数据(即"纯结构"列)."重合度"为使用 Doc2Vec 和 LDA 作为文本算法的检测结果中重复的部分(交集)占全部检测结果(并集)的比重;"非重合部分占比"为去除重复部分的全部检测结果中,二者检测出的信息分别所占的比重.

Table 7 Detection results of coupling design problems (%)

表 7 耦合设计问题的检测结果 (%)

项目名	精确率			召回率			重合度	非重合部分占比	
	Doc2Vec	LDA	纯结构	Doc2Vec	LDA	纯结构	Doc2Vec 和 LDA	Doc2Vec	LDA
Awesome-qr.js	100	100	100	66.67	66.67	66.67	100	–	–
Three.js	61.82	54.65	60.00	77.27	53.41	47.72	76.58	96.15	3.75
PDF.js	70.00	66.67	55.56	87.50	59.38	46.88	57.78	89.47	10.53
FloraJS	66.67	60.00	75.00	83.33	42.86	42.86	55.56	100	0
Pixi.JS	89.66	92.31	92.00	92.86	85.71	82.14	89.66	100	0
Brackets	86.27	87.18	82.50	93.62	72.34	67.35	76.47	100	0

Table 8 Detection results of cohesion design problems (%)

表 8 内聚设计问题的检测结果 (%)

项目名	精确率			召回率			重合度	非重合部分占比	
	Doc2Vec	LDA	纯结构	Doc2Vec	LDA	纯结构	Doc2Vec 和 LDA	Doc2Vec	LDA
Awesome-qr.js	100	100	100	83.33	71.43	57.14	100	–	–
Three.js	74.58	44.44	65.00	71.74	34.28	31.43	17.81	76.67	23.33
PDF.js	65.38	50.00	36.36	76.19	23.81	19.05	35.48	85.00	15.00
FloraJS	75.00	42.86	40.00	75.00	37.50	12.50	36.36	57.14	42.86
Pixi.JS	63.64	40	22.22	87.50	25.00	12.50	44.00	100	0
Brackets	70.73	66.67	55.56	80.56	27.78	16.13	27.27	90.63	9.37

Table 9 Other detection results (%)

表 9 其他检测结果 (%)

项目名	低内聚类占比	高耦合类占比	内聚耦合问题同现率/秩和检验 *P* 值	内聚耦合相关系数/*P* 值	文本信息源占比
Awesome-qr.js	77.78	55.56	55.56%/0.04	–/>0.05	36.36
Three.js	25.43	50.86	66.88%/6.27e–34	0.46/6.3e–18	45.14
PDF.js	5.23	9.91	47.28%/1.34e–08	0.27/0.04	57.06
FloraJS	41.67	54.17	76.92%/1e–04	0.58/0.04	41.38
Pixi.JS	28.41	44.32	48.84%/5.21e–07	0.63/0.002	36.17
Brackets	30.00	46.88	55.70%/6.56e–16	–/>0.05	40.98

4.3 实验讨论

对于 Q1,根据表 7 和表 8 中 Doc2Vec 列的精确率和召回率数据,可见其总体表现与检测 Java Code Smell 的同类文献[15]相仿,JS4C 能够比较准确地检测和识别内聚和耦合问题.如表 7 所示,使用 Doc2Vec 作为文本检测算法的 JS4C 在检测耦合问题方面的整体表现更好,在保持与其他方式相当精确率的前提下,大幅提升了召回率;如表 8 所示,在检测内聚问题时,其优势更为全面和显著.

本文还参考相关文献得出的内聚和耦合问题的关联和性质,通过表 9 中第 4 列、第 5 列的其他指标验证了其中的结论,以印证对 Q1 的回答.Badri 等人[36]指出,内聚和耦合的度量间存在相关性.数据集的 6 个项目中,有 4 个得出了类似结论,其中,Awesome-qr.js 相关性不显著的原因是数据不足.Chahal 等人[8]认为:“高内聚、低耦合”尽管在设计原则中并列出现,但实现高内聚不意味着低耦合,然而在复用性较差的设计中,二者会同时出现.根据秩和检验 *P* 值,可知内聚和耦合问题的分布显著不同,因此它们并不是同一种问题;通过同现率,可印证两种问题有较高概率同时出现.

对于 Q2,JS4C 包含因上游软件和算法等特性较难排除的已知问题,以及针对软件的具体情况而排除的干扰因素,排除干扰因素可以有针对性地提升在特定项目中的表现,已知问题则需要在后续工作中持续改进.

已知问题包括:(1) JS4C 基于类检测工具 JSDeodorant 扩展,检测效果受其制约,尽管 JSDeodorant 是目前效果最好的 JS 类检测工具,但由于实现类的方式多样,工具会漏检部分构造不规范的类(例如使用对象定义且没有构造器的情形),影响到了检测精度;(2) 文本过短会导致文本模型得出不合理的相似度,因此目前有较多代码行数较小的类被误判为低内聚类;(3) 由于 JS 使用弱类型系统,目前使用被调用函数签名的特征推断对象的类,如果特征不明显,对象所属的类可能无法被准确推断,进而影响 NSCDISP,ATFM 等结构方式度量计算.

除此之外,还有能因地制宜而排除的干扰因素.

在检测耦合问题的实验中,JS4C 在 Three.js 和 FloraJS 的例子中的精确率表现不甚理想,主要的错判包括 2 类情况.

(1) 与浏览器支持的 WebGL、Canvas 和页面事件等功能接口交互的类和函数中,JS4C 的结构方式将其识别为一种不明来源的耦合,它们实为对浏览器底层功能的封装.对于这类问题,可以通过建立浏览器底层功能函数库的白名单,并将这些函数列入文本检测流程中的“停用词”范畴,以优化检测效果.

(2) 由于文本相似度检测的特性,FE 的检测会受到 Duplicate Code(重复代码)的干扰.在 Code Smell 优先级排序的相关研究中[37],Duplicate Code 优先于 FE,该问题可通过 Code Smell 排序的方式规避.

在检测内聚问题的实验中,受干扰导致的错判包括 3 类情况.

(1) 数据类(data class)或模型类(model)中与业务逻辑无关的存取和序列化等函数提高了结构方式内聚性的度量值、干扰了判断:一方面,JS 的存取函数并没有通用的模板代码,其命名和实现方式不遵循特定规则;另一方面,部分存取函数也包含少量的业务逻辑,因此不能简单地排除全部的存取函数.

(2) 代码中存在大篇幅的、与业务逻辑无关的变量默认值,影响了文本方式的判断结果.

(3) 部分工具类有低内聚的特性,易被判为低内聚类.对于该问题,一方面,工具类是否为反模式仍有争论;另一方面,若实有必要,可以通过工具函数的检测算法[38]将其排除.

对于 Q3,文本检测算法提升了检测效果:一方面,在表 9 中,易见文本信息源占据了约 4 成的比例,是重要的

信息源;另一方面,文本方式检测出了更多的、有效的信息.根据表 7 和表 8 中 Doc2Vec 和纯结构方式的对比,易见前者使召回率有普遍提升,即检测到了更多的设计问题;Doc2Vec 检测出的额外信息没有对精确率造成严重影响,在内聚问题的检测中还使精确率有显著提升,即检测到了有效的信息.

在文本方式的算法中,Doc2Vec 方法的整体优于 LDA.从表 7 和表 8 中的“重合度”可以看出,两种文本算法的检测结果具有一定相似性;但从“非重合部分占比”数据可以看出,在余下的不重合部分,Doc2Vec 方法普遍比 LDA 能检测出更多不同的数据.根据 Q1 的结论,可见这些数据显著提升了检测效果.因此,Doc2Vec 更适合作为本文的文本分析检测算法.

4.4 有效性威胁

对于 Q4,本文实验评估的有效性威胁包括以下 3 点.

(1) 数据集项目的选取问题.本文从 GitHub 和 JS 类检测的文献[7]中选取了运行环境多样、功能不同且将类作为主要设计模式的 JS 项目用于验证实验结果,但仅涉及了 6 个开源项目,无法涵盖 JS 程序的全部应用领域.

(2) 人工标注设计问题的主观因素.由于缺乏可供参照的同类工作和数据集,因此用于验证有效性的设计问题是人工标注的,存在主观因素.因此,本文参考其他学者[3]的做法,引入具备 JS 开源项目经验的开发者合作标注问题,以期还原现实的软件开发和维护情境.

(3) 动态阈值的选取问题.由于缺乏 JS 类度量指标的统计数据,为了保障检测方法的适应性,本文不使用固定阈值检测.对于 Java Code Smell 的文献中采用固定阈值的度量(例如结构方式的 DC 和 FE),本文根据阈值得出的方式针对每个项目分别统计,形成动态阈值.对于这类改为动态阈值的度量,图 7 展示了全部数据集类中它们原始数值的分布,并用分别标注出了原始文献中的固定值(虚线)和统计数据集得出的值(实线).根据两者差值的绝对值同固定值的比值计算,可以得出不同度量的动态阈值与固定阈值有 5%(LCOM5)~20%(CINT)不等的差异.

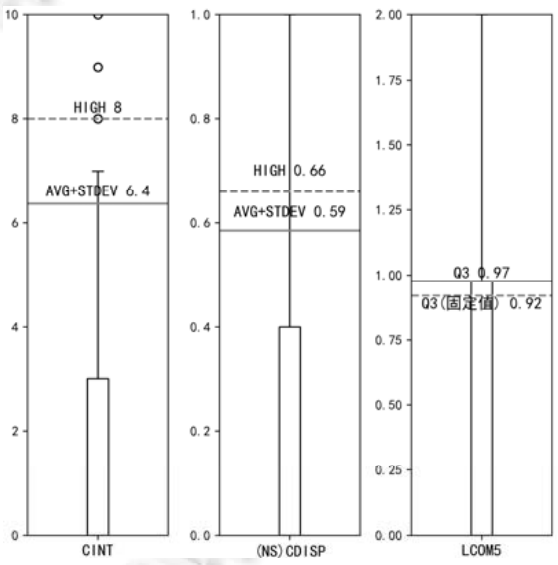

Fig.7 Distributions and thresholds of 3 metrics

图 7 3 种度量的分布及阈值

对于 Q5,JS4C 可以量化代码异味强度,通过提升 JS 类的内聚、降低耦合来消除强度高的代码异味,进而实现高效的 JS 类重构.然而,近期有研究指出:除了异味强度外,还有代码的运行环境、模块和组件的重要程度等情境因素影响代码异味的重构优先级[39],在特殊的情境下,代码异味可能不构成软件设计问题.因此,在实际应用

场景中,未被纳入考虑的情境因素可能降低代码异味强度的参考价值.这一问题可以通过实现情境感知的代码优先级排序来解决,它综合考虑影响代码质量和重构优先级的多方面因素,代码异味强度是其中的一维特征[39].

5 结论和后续工作

JS已成为最常用的编程语言之一,然而在JS项目中,仍未充分实现常见类Code Smell的检测.本文针对DC、FE和Blob这3种Code Smell,结合文本分析和代码结构静态分析,提出了一个检测JS类的内聚耦合Code Smell的方法JS4C,并在实验部分,通过对6个开源项目的分析,证明了JS4C对内聚和耦合的设计问题有良好的检测效果.

后续工作有3个方向:其一,研究的范围可以拓展到基于浏览器端框架的业务代码,检测与框架设计相关的内聚耦合Code Smell;其二,对大量的工业软件项目进行大样本的检测,进一步提高检测工具的稳定性和表现,并对Code Smell及耦合、内聚问题进行详细的质性分析;其三,可以将结构检测和文本检测的结合方法进一步改进,近期出现了基于深度学习模型、抽象语法树中的名称和结构特征及词向量分析代码功能的方法[40],其对Code Smell检测的帮助也是值得研究的.

References:

[1] Fard AM, Mesbah A. Jsnose: Detecting JavaScript Code Smells. In: Proc. of the 13th Int'l Working Conf. on Source Code Analysis and Manipulation. New York: IEEE, 2013. 116−125. [doi: 10.1109/SCAM.2013.6648192]

[2] Ocariza Jr FS, Pattabiraman K, Mesbah A. Detecting unknown inconsistencies in Web applications. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering. New York: IEEE, 2017. 566−577. [doi: 10.1109/ASE.2017.8115667]

[3] Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D. When and why your code starts to smell bad (and whether the smells go away). IEEE Trans. on Software Engineering, 2017,43(11):1063−1088. [doi: 10.1109/TSE.2017.2653105]

[4] Kostanjevec D, Pusnik M, Hericko M, Budimac Z. A preliminary empirical exploration of quality measurement for JavaScript solutions. In: Proc. of the 6th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Aachen (CEUR-WS). 2017. 11−13.

[5] Saboury A, Musavi P, Khomh F, Bostjan S, Gordana R, Zoran B. An empirical study of Code Smells in JavaScript projects. In: Proc. of the 24th Int'l Conf. on Software Analysis, Evolution and Reengineering. New York: IEEE, 2017. 294−305. [doi: 10.1109/SANER.2017.7884630]

[6] Palomba F, Panichella A, Zaidman A, Oliveto R, De Lucia A. The scent of a smell: An extensive comparison between textual and structural smells. IEEE Trans. on Software Engineering, 2017,44(10):977−1000. [doi: 10.1109/TSE.2017.2752171]

[7] Silva LH, Valente MT, Bergel A, Anquetil N, Etien A. Identifying classes in legacy JavaScript code. Journal of Software Evolution and Process, 2017,29(8):Article No.e1864. [doi: 10.1002/smr.1864]

[8] Chahal KK, Singh H. Metrics to study symptoms of bad software designs. ACM SIGSOFT Software Engineering Notes, 2009,34(1):1−4. [doi: 10.1145/1457516.1457522]

[9] Fowler M, Beck K. Refactoring: Improving the Design of Existing Code. 2nd ed., Boston: Addison-Wesley Professional, 2019.

[10] Fowler M. Refactoring a JavaScript video store. 2020. https://martinfowler.com/articles/refactoring-video-store-js/

[11] Inheritance and the prototype chain. 2020. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

[12] JS Classes. 2020. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

[13] Rostami S, Eshkevari L, Mazinanian D, Tsantalis N. Detecting function constructors in JavaScript. In: Proc. of the Int'l Conf. on Software Maintenance and Evolution. New York: IEEE, 2016. 488−492. [doi: 10.1109/ICSME.2016.29]

[14] Lanza M, Marinescu R. Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems. Berlin: Springer Science and Business Media, 2007.

[15] Palomba F, Panichella A, De Lucia A, Oliveto R, Zaidman A. A textual-based technique for smell detection. In: Proc. of the 24th Int'l Conf. on Program Comprehension. New York: IEEE, 2016. 1−10. [doi: 10.1109/ICPC.2016.7503704]

[16] Palomba F, Zanoni M, Fontana FA, De Lucia A, Oliveto R. Toward a smell-aware bug prediction model. IEEE Trans. on Software Engineering, 2019,45(2):194−218. [doi: 10.1109/TSE.2017.2770122]

[17] Fontana FA, Ferme V, Zanoni M, Yamashita A. Automatic metric thresholds derivation for Code Smell detection. In: Proc. of the 6th Int'l Workshop on Emerging Trends in Software Metrics. New York: IEEE, 2015. 44−53. [doi: 10.1109/WETSoM.2015.14]

[18] Palomba F, Oliveto R, De Lucia A. Investigating Code Smell co-occurrences using association rule learning: A replicated study. In: Proc. of the IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation. New York: IEEE, 2017. 8−13. [doi: 10.1109/MALTESQUE.2017.7882010]

[19] Lin T, Gao JH, Fu X, Lin Y. A container-destructor-explorer paradigm to Code Smells detection. Journal of Chinese Computing Systems, 2016,37(3):469−473 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-1220.2016.03.014]

[20] Fokaefs M, Tsantalis N, Chatzigeorgiou A. Jdeodorant: Identification and removal of feature envy bad smells. In: Proc. of the 23rd Int'l Conf. on Software Maintenance. New York: IEEE, 2007. 519−520. [doi: 10.1109/ICSM.2007.4362679]

[21] Blei DM, Ng AY, Jordan MI. Latent dirichlet allocation. Journal of Machine Learning Research, 2003,3:993−1022. [doi: 10.1162/jmlr.2003.3.4-5.993]

[22] Bavota G, Oliveto R, De Lucia A, Marcus A, Gueheneuc YG, Antoniol G. In medio stat virtus: Extract class refactoring through Nash equilibria. In: Proc. of the IEEE Conf. on Software Maintenance, Reengineering, and Reverse Engineering. New York: IEEE, 2014. 214−223.[doi: 10.1109/CSMR-WCRE.2014.6747173]

[23] Le Q, Mikolov T. Distributed representations of sentences and documents. In: Proc. of the Int'l Conf. on Machine Learning. Cambridge: JMLR, 2014. 1188−1196. [doi: 10.5555/3044805.3045025]

[24] Closure compiler. 2020. https://developers.google.com/closure/compiler/

[25] Nicolay J, Noguera C, De Roover C, De Meuter W. Determining dynamic coupling in JavaScript using object type inference. In: Proc. of the 13th Int'l Working Conf. on Source Code Analysis and Manipulation. New York: IEEE, 2013. 126−135. [doi: 10.1109/SCAM.2013.6648193]

[26] Moha N, Gueheneuc YG, Duchien AF. Decor: A method for the specification and detection of code and design smells. IEEE Trans. on Software Engineering, 2010,36(1):20−36. [doi: 10.1109/TSE.2009.50]

[27] Al Dallal J. Measuring the discriminative power of object-oriented class cohesion metrics. IEEE Trans. on Software Engineering, 2011,37(6):788−804. [doi: 10.1109/tse.2010.97]

[28] Aji S, Kaimal R. Document summarization using positive pointwise mutual information. Int'l Journal of Computer Science and Information Technology, 2012,4(2):Article No.47. [doi: 10.5121/ijcsit.2012.4204]

[29] Niwa Y, Nitta Y. Co-Occurrence vectors from corpora vs. distance vectors from dictionaries. In: Proc. of the 15th Conf. on Computational Linguistics. ACL, 1994. 304−309. [doi: 10.3115/991886.991938]

[30] Kuhn A, Ducasse S, Gírba T. Semantic clustering: Identifying topics in source code. Information and Software Technology, 2007, 49(3):230−243. [doi: j.infsof.2006.10.017]

[31] Han JM, Wang W, Li T, He Y. Feature location method of evolved software. Journal of Frontiers of Computer Science and Technology, 2016,10(9):1201−1210 (in Chinese with English abstract). [doi: j.issn.1673-9418.1507062]

[32] Awesome-qr.js. 2020. https://github.com/SumiMakito/Awesome-qr.js

[33] three.js. 2020. https://github.com/mrdoob/three.js

[34] Zar JH. Significance testing of the Spearman rank correlation coefficient. Journal of the American Statistical Association, 1972, 67(339):578−580. [doi: 10.1080/01621459.1972.10481251]

[35] Wilcoxon F. Individual comparisons of grouped data by ranking methods. Journal of Economic Entomology, 1946,39(2):269−270. [doi: 10.1093/jee/39.2.269]

[36] Badri L, Badri M, Touré F. Exploring empirically the relationship between lack of cohesion in object-oriented systems and coupling and size. In: Proc. of the 5th Int'l Conf. on Software and Data Technologies. Setubal: SciTe Press, 2010. 317−324. [doi: 10.1007/978-3-642-17578-7_9]

[37] Gao Y, Liu H, Fan XZ, Niu ZD, Shao WZ. Resolution sequence of bad smells. Ruan Jian Xue Bao/Journal of Software, 2012,23(8): 1965−1977 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4152.htm [doi: 10.3724/SP.J.1001.2012.04152]

[38] Mendes T, Valente MT, Hora A. Identifying utility functions in Java and JavaScript. In: Proc. of the Brazilian Symp. on Software Components, Architectures and Reuse. New York: IEEE, 2016. 121−130. [doi: 10.1109/SBCARS.2016.16]

[39] Sae-Lim N, Hayashi S, Saeki M. Context-Based Code Smells prioritization for prefactoring. In: Proc. of the 24th Int'l Conf. on Program Comprehension. New York: IEEE, 2016. 1−10. [doi: 10.1109/icpc.2016.7503705]

[40] Alon U, Zilberstein M, Levy O, Yahav E. code2vec: Learning distributed representations of code. Proc. of the ACM on Programming Languages (POPL), 2019,3:Article No.40. [doi: 10.1145/3290353]

附中文参考文献:

[19] 林涛,高建华,伏雪,林艳.面向 Code Smells 的"容器-破坏者-发现者"检测策略.小型微型计算机系统,2016,37(3):469−473. [doi: 10.3969/j.issn.1000-1220.2016.03.014]

[31] 韩俊明,王炜,李彤,何云.演化软件的特征定位方法.计算机科学与探索,2016,10(9):1201−1210. [doi: j.issn.1673-9418.1507062]

[37] 高原,刘辉,樊孝忠,牛振东,邵维忠.代码坏味的处理顺序.软件学报,2012,23(8):1965−1977. http://www.jos.org.cn/1000-9825/4152.htm [doi: 10.3724/SP.J.1001.2012.04152]

黄子杰(1994−),男,博士生,CCF 学生会员,主要研究领域为代码异味,软件可靠性,实证软件工程.

高建华(1968−),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件重构,软件测试,可信软件,可靠性模型设计.

陈军华(1968−),男,副教授,主要研究领域为数据库理论及应用,分布式数据库.