基于分层架构模式识别的软件架构重构技术

王 丽[1,2],杜鹏程[1],许一鸣[2],李必信[1]
(1. 东南大学计算机科学与工程学院,江苏南京 210096; 2. 江苏自动化研究所,江苏连云港 222061)

摘 要: 本文提出一种基于分层架构模式识别的软件架构重构技术. 该技术以目标软件的源代码作为输入,过滤与分层架构无关的代码,再利用代码词汇信息挖掘程序实体之间的语义关联,通过代码主题提取并计算程序实体之间的职责相似度,依据相似度将程序实体聚类形成组件. 在软件组件化的基础上结合分层模式的 ILD 属性识别软件层次和软件架构模式. 在模式识别的基础上,定位系统中存在的违规作为重构点,生成相应的重构建议并实施重构. 最后,本文在 Github 与 SourceForge 开源社区中选取 10 个开源软件系统作为实验对象,验证了本文提出的基于分层架构模式识别的软件架构重构技术在模式识别有效性、重构点识别效果和重构建议实施效果方面与传统方法相比有较大提升,能够有效的帮助软件开发人员识别软件架构模式、获取重构点、生成重构建议,并协助开发人员进行架构重构实施,改善系统违规情况,提升软件质量.

关键词: 架构重构; 分层架构模式; 架构模式识别; 重构点定位
中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112 (2021)01-0201-08
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.12263/DZXB.20191198

Software Architecture Reconstruction Technology Based on Layered Architecture Pattern Recognition

WANG Li[1,2], DU Peng-cheng[1], XU Yi-ming[2], LI Bi-xin[1]
(1. *School of Computer Science and Engineering, Southeast University, Nanjing, Jiangsu* 210096, *China*;
2. *Jiangsu Automation Research Institute, Lianyungang, Jiangsu* 222061, *China*)

Abstract: This paper proposes software architecture reconstruction technology based on layered architecture pattern recognition. The input of the recognition is the source code and the unnecessary source code will be filtered out at first. Then the approach relies on lexical information from the source code to mine the semantic relation between system entities and using a topic model to extract the responsibility of entities, which is then used to cluster these entities into cohesion components. Later, the approach supplements the structural information between components to generate the component graph and use the ILD property to recognize the actual software layers. Based on the results of pattern recognition and the principle of layered pattern, position the nonstandard existing in the system as the reconstruction point, and relevant reconstruction suggestions to assist the designers and developers in the reconstruction implementation. Finally, this paper selects 10 open source software systems in Github and SourceForge as experimental objects to verify the effectiveness of the technology in this paper. This technology can greatly improve the effectiveness of pattern recognition, the recognition effect of illegal refactoring points and the implementation effect of refactoring suggestions. This technology can also assist developers in the implementation of architecture reconstruction to a certain extent, improve the situation of the system violations, and improve the quality of the software.

Key words: architecture refactoring; layered pattern; recognition of architectural patterns; refactoring point positioning

1 引言

在软件系统不断演进过程中原始架构常常出现架构偏移与腐蚀现象,导致系统实际架构与原始架构不一致[1]. 此外,在软件更新过程中架构信息的丢失,使得软件实际架构与设计架构之间偏差越来越大,造成系统维护成本不断攀升,软件质量不断恶化[2]. 因此,研究软件架构识别与重构技术具有重要意义.

分层架构模式是目前最常用的软件架构模式之一,这种软件架构模式能够降低系统复杂度,提高系统

收稿日期:2019-10-22;修回日期:2020-04-05;责任编辑:孙瑶
基金项目:国家自然科学基金(No. 61572126, No. 61872078);"十三五"装备预研共用技术(No. 41402020501)

可修改性、可重用性、可移植性,减小系统耦合度,易于维护[3~5]. 在分层架构模式识别方面,Müller 等人提出了一种创建目标系统高等抽象的表示方法,帮助用户发现与构建目标系统的子系统结构[6,7]. Andreopoulos 等人提出了一种新的聚类算法 MULIsoft,该算法结合目标系统静态信息与运行动态信息恢复目标系统的分层架构[8]. 但上述方法识别获取的分层架构与目标系统分层架构重合度不高,识别精度较低. Laval 等人[9]提出了一种启发式的遗传算法,该算法能够在系统依赖结构存在循环依赖的情况下将目标系统的包结构分解为分层结构,但是未给出实用的重构实施建议. 综合上述问题可以看出,目前业界仍缺乏精度较高的分层架构模式识别方法和基于识别结果的重构实施研究.

本文针对分层软件架构模式特点,获取软件的分层结构信息以及不同层次间的依赖关系,采用逆向工程方法还原软件架构,并根据分层架构使用规范检测违反分层架构的设计并定位重构点,根据重构点生成重构建议.

本文的创新点主要包括:①提出一种基于软件职责和结构的分层架构模式识别方法,提升了模式识别结果的准确性,使得模式识别结果更接近系统的实际分层架构;②提出针对分层架构模式使用规则的形式化的重构点定位方法,通过合规性判定准确查找软件中的重构点;③针对三种违规类型的重构点,生成对应的架构重构建议,并将架构重构建议映射成为代码层面重构建议,有效指导设计开发人员开展软件架构重构.

2 分层架构模式识别方法

分层架构模式识别方法以目标系统的软件源代码为对象,通过预处理过滤与分层架构无关的代码,减少代码噪声对于识别精度的影响,提高识别的准确性[10];采用 JDT(Eclipse Java Development Tools)对目标系统的源代码进行解析[11,12],提取代码词汇信息和主题词信息,通过组件化构建目标系统分层架构;在组件化的基础上补全组件之间的结构依赖信息生成完整的组件图,采用 ILD(Incremental Layer Dependency)属性进行层次识别[16]得到目标系统的分层结构. 最后,对识别结果进行调优,生成最终识别结果.

2.1 代码预处理

代码中与系统架构无关的信息会严重干扰软件架构模式识别过程,影响识别精度和效率,本文采用通用规则、领域规则以及自定义规则对目标系统的源代码进行过滤,具体过程如图 1 所示.

2.2 软件职责恢复

本文采用代码词汇信息提取、主题词提取以及组件化三个步骤来分析软件实体之间的职责相似程度,将其聚类生成代码职责组件.

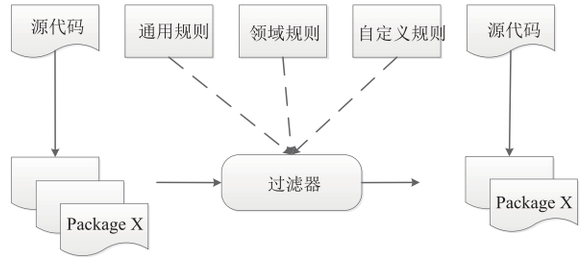

图1 代码预处理过程

(1)代码词汇信息提取

将预处理后的源代码类文件转换为抽象语法树(AST,Abstract Syntax Tree),通过遍历 AST 的节点提取出每个类文件中的词汇信息. 由于软件职责的程序实体聚类层级为系统的 package 层,需要将每个类文件提取的词汇按照其所属 package 进行合并,每个 package 对应生成一个词汇文档.

(2)代码主题词提取

采用分词法将 package 词汇文档中的合成词切分成独立的单词,通过词干化提取所有单词的词干或者词根,再去除不能传递有效信息的单词来消除噪声. 不同位置词汇的重要性不同,可以通过出现次数来对词汇进行加权处理:

$$\mathrm{Occurence}(w_i,D_p)=\mathrm{requecy}(w_i,D_p)*\mathrm{eight}(it_{wi})$$

其中,$\mathrm{Occurence}(w_i,D_p)$ 表示单词 w_i 在加权后在 package$p$ 的文档集 D_p 中出现次数,$\mathrm{frequency}(w_i,D_p)$ 表示单词 w_i 在文档集 D_p 中原本出现的频率,it_{wi}表示单词 w_i 在源代码中来源的位置,$\mathrm{weight}(it_{wi})$ 是词汇的权重.

采用文档主题生成模型(LDA,Latent Dirichlet Allocation)对代码词汇文档进行主题词提取,提取的内容包括[13,14]:代码词汇文档集的主题 $T=\{t_1,t_2,\cdots,t_n\}$,n 为主题数量,t_n 为分布概率;文档主题分布向量 $\boldsymbol{V}_p=\{\bar{v}_1,\bar{v}_2,\cdots,\bar{v}_m\}$,$m$ 为 package 数目,$\bar{v}_i$ 对应第 i 个 package 的主题分布向量,$\bar{v}_i$ 是该 package 的显著分布程度. 将 $\boldsymbol{V}_p$ 作为代表 package 职责相关的特征向量,$\boldsymbol{V}_p$ 表示其程序实体语义相关的内涵.

(3)组件化

本文在 package 粒度基础上将职责相似的 package 聚类到相同的组件中. 聚类过程建立在 SSP(Semantic Similarity Between Packages)[15]度量值的基础之上:

$$\mathrm{SSP}(p_i,p_j)=\frac{\bar{v}_i\cdot\bar{v}_j}{\|\bar{v}_i\|\cdot\|\bar{v}_j\|} \tag{1}$$

其中,$\bar{v}_i$ 和 $\bar{v}_j$ 是package$i$ 和package$j$ 的主题分布向量,$\mathrm{SSP}(p_i,p_j)$ 越大,package$i$ 和package$j$ 的职责相似度

越高.

基于软件职责的聚类结果需要保证组件内 SSP 耦合度较高,不同组件之间 SSP 耦合度较低,$\eta(C_i)$代表组件 C_i 内部 SSP 耦合度[15]:

$$\eta(C_i) = \frac{2 * \sum_{s=1}^{n} \sum_{k=1, s\neq k}^{n} \mathrm{SSP}(P_s, P_k)}{n(n-1)} \tag{2}$$

组件 C_i 与组件 C_j 之间 SSP 耦合度为

$$\delta(C_i, C_j) = \frac{\sum_{P_s \in C_i} \sum_{P_k \in C_j} \mathrm{SSP}(P_s, P_k)}{n * m} \tag{3}$$

其中,n 与 m 分别为组件 C_i 和 C_j 中独立 package 的数量.

适应性函数 RRQ(Responsibility Recovery Quality)[15]能够提升组件内部 SSP 耦合度并降低组件之间 SSP 耦合程度. 利用 RRQ 能够将基于软件职责的组件化问题化为寻找一个最佳的 package 组合方式的问题.

SCF(C_i)(Semantic Component Factor)为组件 C_i 内部 SSP 耦合值 $\eta(C_i)$与 C_i 之外的组件之间所有耦合度 $\delta_{i\neq j}(C_i, C_j)$之和的归一化比值[7,15]:

$$\mathrm{SCF}(C_i) = \begin{cases} 0, & \eta_i(C_i) = 0 \\ \dfrac{\eta_i}{\eta_i(C_i) + \sum_{j=1, j\neq i}^{k} \delta(C_i, C_j)}, & \eta_i(C_i) \neq 0 \end{cases} \tag{4}$$

RRQ 值等于所有组件的 SCF 值累加,k 的值为聚类的中间过程中某一状态下生成的所有组件的数目:

$$\mathrm{RRQ} = \sum_{i=1}^{k} \mathrm{SCF}(C_i) \tag{5}$$

2.3　软件层次识别

软件层次识别是在软件组件化的基础上补全组件之间的结构依赖信息,生成完整的组件图,并结合分层架构模式的 ILD 属性进行层次识别[16],最终得到目标系统的分层结构.

首先,将目标系统的类文件转换为 AST 形式,通过遍历 AST 的节点提取所需类之间的结构依赖信息,最后生成类之间的结构依赖图. 定义类结构依赖图 $G_{\mathrm{cls}} < \mathrm{clses}, \mathrm{Edge}_{\mathrm{cls}} >$ 由类节点集合 clses 以及类之间依赖关系的边集合$\mathrm{Edge}_{\mathrm{cls}}$构成. G_{cls}使用邻接表进行抽象表示,将目标系统中实体类 i 生成一个对应的顶点cls_i,将所有实体类顶点存放在一维数组中,同时以这些顶点作为依赖关系的起点,记录每个类顶点的指向的依赖类,如图 2 所示.

图 2(a)表示类结构依赖图,每个点代表系统中对应的一个实体类,每条边代表类对类之间的一种依赖关系;图 2(b)代表描述类结构依赖信息的邻接矩阵.

由于组件依赖图生成需要 package 之间的结构依赖信息,但是类结构依赖图仅仅包含类之间的关系,所以生成组件依赖图还需要进行 package 结构依赖图生成. 如图 3 所示,package A 对 package B 的依赖源自 package A 中包含的类对 package B 中包含的类之间的依赖,类之间的依赖边可由多种依赖构成.

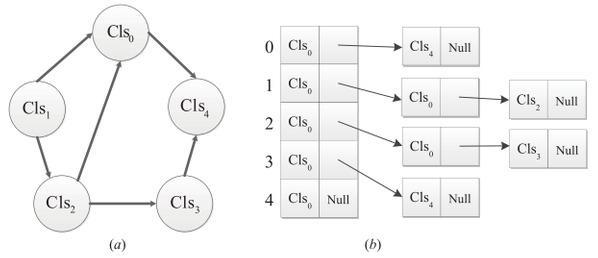

图2　类结构依赖图数据结构

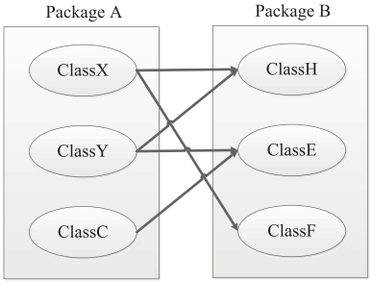

图3　package之间结构依赖

遍历一个类对另外一个类的所有依赖类型,并对其进行加权统计就可以获得 package 之间的结构依赖强度. 类 A 和类 B 之间依赖关系包括继承、实现、组合、参数传递、实例化以及返回类型六种,不同依赖关系的权重如表 1 所示.

表 1　类之间依赖对应权重表

依赖关系	继承	实现	组合	参数传递	实例化	返回类型
权重	5	5	3	1	2	1

类 A 与类 B 之间的结构依赖强度 W_{cls}是其依赖类型的加权和:

$$W_{\mathrm{cls}}(\mathrm{CLS_A}, \mathrm{CLS_B}) = \sum_{\mathrm{Edge}_{\mathrm{cls}}} \mathrm{weightage}(\mathrm{Edge}_{\mathrm{cls}}) \tag{6}$$

package A 到 package B 之间的依赖强度 W_{pkg}可通过累计 package A 到 package B 的类结构依赖强度得到:

$$W_{\mathrm{pkg}}(P_{\mathrm{A}}, P_{\mathrm{B}}) = \sum_{\mathrm{CLS_A} \in P_{\mathrm{A}}, \mathrm{CLS_B} \in P_{\mathrm{B}}} W_{\mathrm{CLS}}(\mathrm{CLS_A}, \mathrm{CLS_B}) \tag{7}$$

计算组件 A 和组件 B 之间的依赖关系 W_C 需要在 $G_{\mathrm{pkg}} < \mathrm{pkgs}, \mathrm{Edge}_{\mathrm{pkgs}} >$ 中遍历组件 A 中的 package,如果 A 的邻接节点是组件 B 中的 package,则将组件 A 与组件 B 的依赖强度增加相应 package 之间的依赖权值:

$$W_C(C_{\mathrm{A}}, C_{\mathrm{B}}) = \sum_{P_i \in C_{\mathrm{A}}, P_j \in C_{\mathrm{B}}} W_{\mathrm{pkg}}(P_i, P_j) \tag{8}$$

对组件之间的加权依赖求和可以得到组件依赖图 $G_C < \mathrm{Components}, \mathrm{Edge}_C >$. 组件依赖图 G_C 中目标系

统的分层结构 LayeredSystem = $\{\text{layer}_1, \text{layer}_2, \cdots, \text{layer}_n\}$ 的任意层次layer_i由任意数量的组件组成. 分层架构可以抽象出四种类型的依赖[17]:相邻层间依赖(AdjacentCall)、层内依赖(IntraCall)、跨层依赖(SkipCall)和反向依赖(BackCall). 四种类型的依赖度量分别为[17]:

$$\begin{aligned}&\text{AdjacentCall}(\text{layer}_i,\text{layer}_j)\\&=\sum_{C_A\in\text{layer}_i,C_B\in\text{layer}_j}W_C(C_A,C_B);i=j+1\\&\text{IntraCall}(\text{layer}_i)\\&=\sum_{C_A\in\text{layer}_i,C_B\in\text{layer}_i,C_A\neq C_B}W_C(C_A,C_B)\\&\text{SkipCall}(\text{layer}_i,\text{layer}_j)\\&=\sum_{C_A\in\text{layer}_i,C_B\in\text{layer}_j}W_C(C_A,C_B);j<i-1\\&\text{BackCall}(\text{layer}_i,\text{layer}_j)\\&=\sum_{C_A\in\text{layer}_i,C_B\in\text{layer}_j}W_C(C_A,C_B);i<j\end{aligned}$$

分层架构应该满足如下的约束条件[17]:

$$\begin{aligned}&\forall i,j,k(j<i,k<j-1)\\&\Rightarrow\text{BackCall}(\text{layer}_j,\text{layer}_i)\\&<\text{SkipCall}(\text{layer}_j,\text{layer}_k)\\&\quad\text{AdjacentCall}(\text{layer}_j,\text{layer}_{j-1})\end{aligned}\tag{9}$$

即系统中的任意层次对上层实体的反向依赖数量应该小于其对非相邻下层的依赖数量,同时远远小于其对相邻下层的依赖数量. 此外,目标系统的分层架构还需要满足如果层次内部组件之间的依赖相对较少,只有在系统某些属性需要提升的要求下才有可能加强层次内部依赖强度的约束条件[17]:

$$\forall i,\text{IntraCall}(\text{layer}_i)\ \text{AdjacentCall}(\text{layer}_i,\text{layer}_{i-1})\tag{10}$$

依据上述约束条件,将组件分配到一组软件层次中,并且满足约束的要求:①奖励层次间的相邻依赖;②保持层内依赖处于较低水平;③最小化跨层依赖以及反向依赖. 对任一层次layer_i引入代价函数 ILC(Individual Layering Cost)[16]:

$$\begin{aligned}\text{ILC}(i)=&ap*\text{AdjacentCall}(\text{layer}_i,\text{layer}_{i-1})\\&+ip*\text{IntraCall}(\text{layer}_i)\\&+sp*\sum_{j=i-2}^{1}\text{SkipCall}(\text{layer}_i,\text{layer}_j)\\&+bp*\sum_{j=i+2}^{n}\text{BackCall}(\text{layer}_i,\text{layer}_j)\end{aligned}\tag{11}$$

其中,n是目标系统假定层次划分所包含的层次数目,ap、ip、sp和bp分别对应于层次间相邻依赖、层内依赖、跨层依赖和反向依赖的惩罚因子. sp和bp的值应远大于ap和ip的值. 全局的层代价 LC 如式(12)[16]:

$$LC=\sum_{i=1}^{n}\text{ILC}(\text{layer}_i)\tag{12}$$

LC 值越低,代表层次划分方式越好. 将层次划分问题转化为寻找最小 LC 值问题.

2.4 软件分层架构模式识别结果优化

从每一层次包含的类出发进行优化,遍历分层架构任意层次中所有类,通过查找类结构依赖图 G_{cls} 中生成该类所依赖的类列表,然后对于所有被依赖的类检查其所处层次以及两个类之间的依赖类型是否符合要求. 如果该类与被依赖的类处在上下相邻的层次,并且两者之间是继承关系或者实现关系,则将该类所属的 package 以一个单独组件的形式移动到下一层次中,同时更新组件结构依赖图 G_C 实现模式识别结果优化.

3 基于模式识别结果的分层软件架构重构

软件架构重构是通过改进设计来提高软件质量[18]. 重构的核心在于重构点定位、重构建议生成、重构实施和效果评估. 重构点定位和重构建议生成是架构重构的难点. 本文提出基于分层架构模式识别的架构重构流程,如图 4 所示.

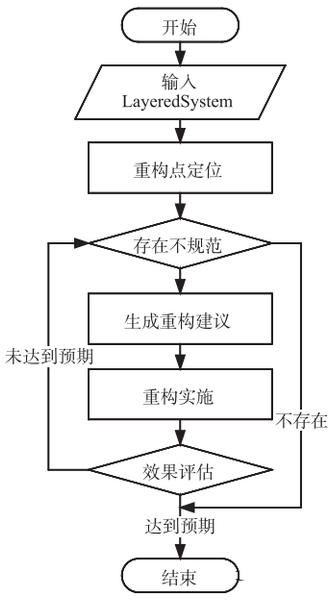

图4 软件架构重构流程

3.1 基于模式识别的重构点定位

本文采用模式合规判定将三种违规的形式化表示作为重构点. SKIP、BACK、INTERFACE 分别是违反相邻关联原则、单向关联原则与面向接口原则的程序实体,三种违规的定位过程形式化定义如下:

$$\mathrm{SKIP}=\{\mathrm{pkg_A}\mid\exists\,\mathrm{Edge_{pkg}}(\mathrm{pkg_A},\mathrm{pkg_B}),\mathrm{pkg_A}\in\mathrm{layer}_i,\mathrm{pkg_B}\in\mathrm{layer}_j,i<j+1\}\tag{13}$$

$$\mathrm{BACK}=\{\mathrm{pkg_A}\mid\exists\,\mathrm{Edge_{pkg}}(\mathrm{pkg_A},\mathrm{pkg_B}),\mathrm{pkg_A}\in\mathrm{layer}_i,\mathrm{pkg_B}\in\mathrm{layer}_j,i<j\}\tag{14}$$

$$\mathrm{INTERFACE}=\left\{\begin{matrix}\mathrm{pkg_A}\mid\exists\,\mathrm{Edge_{cls}}(\mathrm{cls}_X,\mathrm{cls}_Y),\mathrm{cls}_X\in\mathrm{pkg_B},\mathrm{cls}_Y\in\mathrm{pkg_A},\\ \mathrm{pkg_A}\in\mathrm{layer}_i,\mathrm{pkg_A}\in\mathrm{layer}_{i+1},\mathrm{cls}_Y\neq\mathrm{interface}\end{matrix}\right\}\tag{15}$$

SKIP 与 BACK 是从 package 结构依赖图出发,分别定位违规使用其非相邻下层与上层服务的 package,将其视为分层违规依赖的起点. INTERFACE 则是从底层的类结构依赖图出发,如果 package B 中存在实体类 cls_X 依赖其下层中 package A 的实体类cls_Y,并且类cls_Y并不是接口形式,而是直接的功能实现类,则认为 package A 存在非面相接口违规.

3.2　重构建议生成

本文在重构点定位基础上遵循自上而下原则,面向架构重构及代码重构两个层面提出架构重构建议. 如图 5 所示,在分层架构中 P_A违规调用了非直接相邻下层实体 P_B的功能,因而在重构点定位步骤中 P_A被添加进了 SKIP 集合中. 为了使目标系统符合分层模式使用规范,提出切断 P_A到 P_B依赖边的重构建议. 通过进一步分析,发现 P_A到 P_B的依赖衍生于其内部类文件之间的依赖,因此将切断 P_A到 P_B的依赖这一条重构建议映射到代码层面,衍生出消除$\{C_1\rightarrow C_5,C_3\rightarrow C_4,C_3\rightarrow C_5,C_2\rightarrow C_4\}$四条类之间依赖边的重构建议.

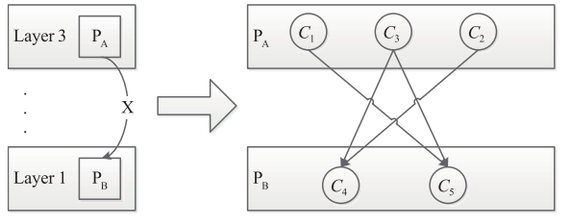

图5　架构重构映射到代码重构

根据三种分层违例的特点,将架构重构建议分为两种类型:切断 package 之间违规依赖和调整 package 内部实体接口. 将代码层面的重构建议同样分为对应两种类型:切断类之间违规依赖和抽象实体类被依赖的部分为接口.

3.3　重构实施与效果评估

为评估重构实施后的效果,需要引入相应的度量指标,包括跨层违规相关度量、反向违规相关度量和非面向接口违规相关度量.

(1)跨层违规相关度量

SCVL(Skip Call Violation in a Layer)是给定层次的跨层违规程度[19]:

$$\mathrm{SCVL}(\mathrm{layer}_i)=\frac{|\{\mathrm{pkg}\in\mathrm{SKIP}\mid\mathrm{pkg}\in\mathrm{layer}_i\}|}{|\{\mathrm{pkg}\in\mathrm{layer}_i\}|}\tag{16}$$

SCVL 用违规跨层调用的 package 数目除和整个层次中的 package 数目的比值评估跨层违规程度.

SCV 是整个目标系统对应跨层违规的合规程度[19]:

$$\mathrm{SCV}(\mathrm{layeredSystem})=\begin{cases}1-\dfrac{\sum_{i=3}^{n}\mathrm{SCVL}(\mathrm{layer}_i)}{n-2}, & \mathrm{if}(n>2)\\ 1, & \mathrm{otherwise}\end{cases}\tag{17}$$

SCV 取值范围是[0,1],SCV 取值越高该分层结构越符合规定.

(2)反向违规相关度量

BCVL(Back Call Violation in a Layer)是给定某个层次的反向违规程度[19]:

$$\mathrm{BCVL}(\mathrm{layer}_i)=\frac{|\{\mathrm{pkg}\in\mathrm{SKIP}\mid\mathrm{pkg}\in\mathrm{layer}_i\}|}{|\{\mathrm{pkg}\in\mathrm{layer}_i\}|}\tag{18}$$

BCVL 用违规调用上层功能的 package 数除以层次中所有 package 数表示反向违规程度.

BCV 是整个目标系统反向违规的合规程度度量[19]:

$$\mathrm{BCV}(\mathrm{layeredSystem})=\begin{cases}1-\dfrac{\sum_{i=1}^{n-1}\mathrm{BCVL}(\mathrm{layer}_i)}{n-1}, & \mathrm{if}(n>1)\\ 1, & \mathrm{otherwise}\end{cases}\tag{19}$$

BCV 取值范围是[0,1],BCV 取值越高反向违规的合规程度越高.

(3)非面向接口违规相关度量

NIOVL(Not Interface Oriented Violation in a Layer)是非面向接口实现违规程度:

$$\mathrm{NIOVL}(i)=\frac{|\{\mathrm{pkg}\in\mathrm{BACK}\mid\mathrm{pkg}\in\mathrm{layer}_i\}|}{|\{\mathrm{pkg}_{\mathrm{depended}}\in\mathrm{layer}_i\}|}\tag{20}$$

NIOV 是整个目标系统的非面向接口合规程度:

$$\mathrm{NIOV}(\mathrm{layeredSystem})=\begin{cases}1-\dfrac{\sum_{i=1}^{n-1}\mathrm{NIOVL}(\mathrm{layer}_i)}{n-1}, & \mathrm{if}(n>1)\\ 1, & \mathrm{otherwise}\end{cases}\tag{21}$$

NIOV 取值范围是[0,1],NIOV 取值越高该系统的非面向接口合规程度越高.

4　实验与分析

本文从模式识别有效性、违规重构点识别效果和

重构建议实施效果三个方面验证基于分层架构模式识别的软件架构重构技术的有效性.

实验环境的硬件平台为 IBM x3850 x6,CPU 型号为 Intel Xeon CPU E7-4809 v2 1.90GHz,运行内存32GB,硬盘320GB,操作系统为 Windows Server 2008 R2 Enterprise SP1(64 bit).实验运行过程涉及到的软件系统包括开发环境 jdk-1.8.0_45 Intellij IDEA,数据库环境 MongoDB 3.6.2 Mysql-5.6.24,web 运行环境 Tomcat 8.0.36、Python 运行环境 Anaconda Python3.6.3 以及第三方库 Nltk Gensim.

在 Github 与 SourceForge 开源社区中选取了 10 个流行度较高的分层架构模式开源软件系统作为实验对象,如表 2 所示.

表 2 开源项目信息

项目名称	代码规模	软件功能
Spring-petclinic	2K	Spring 项目组样例
MyBlog	4K	Java web 开源框架实现的博客系统
Junit3	4K	一款 Java 程序的单元测试框架
Solo	3K	多个 Java 开源框架的博客系统
Symphony	7.5k	Java 实现的下一代社区系统平台
Blog	3K	基于 spring,hibernate,struts 的个人博客
Latke	18k	以 JSON 为主的 Java web 框架
Ssm-master	2k	基于 SSM 框架实现的样例
JHotdraw	32k	开源 Java 图形软件
JFreeChart	133k	开源 Java 图表插件

4.1 验证目的一:模式识别有效性

本实验将基于分层架构模式识别的软件架构重构技术实现为分层架构模式识别及辅助重构工具 LARRT,并对实验对象进行分层架构模式识别. 利用已有的基于结构的识别方法 SI(Structural Information only)和结合了代码词汇的层次识别方法 LSI(Lexical and Structural Information)[17] 对实验对象进行分层架构模式识别. 通过人工方式对实验对象进行层次划分获取权威的软件分层架构. 比较 LARTT、SI 和 LSI 三种方法的识别结果与权威分层架构的重合度,并分别计算每种方法的正确性、完整性以及两者的调和平均值的度量值,如表 3 所示.

从表 3 可以看出,LARTT 识别的正确性和完整性相比较另外两种方法的结果普遍有较大的提升. 对调和平均值而言,LARTT 识别结果在 spring-petclinc 项目上取得了最高值 1.0,在 JFreeChart 项目上所有方法的调和平均值度量值都呈现了较低水平,在 Latke 项目上三种方法的识别效果也相对较差. 相较已有的方法,LARTT 识别结果的调和平均值值达到了 0.7415,这也意味着 LARRT 识别结果与目标系统的权威层次架构重合度更高,识别有效性更高,能够为设计开发人员理解软件架构提供真实有效的帮助.

4.2 验证目的二:违规重构点识别效果

使用 LARRT 对实验对象进行重构点定位,结果如表 4 所示.

通过人工查看目标系统,确认所有重构点都与实际问题相符. 但是存在漏报的重构点,经过分析主要原因是模式识别结果的误差导致. 实验表明,通过 LARTT 识别获取的重构点能够准确定位软件的设计问题,但存在一定的漏报现象.

表 3 LARRT 对开源项目架构识别的有效性对比

项目名称	正确性			完整性			调和平均值		
	LARRT	SI	LSI	LARRT	SI	BM	LARRT	SI	LSI
Spring-petclinic	1.0	0.615	0.718	1.0	0.615	0.718	1.0	0.615	0.718
MyBlog	0.78	0.54	0.67	0.78	0.54	0.67	0.78	0.54	0.67
Junit3	0.734	0.723	0.65	0.734	0.723	0.65	0.734	0.723	0.65
Solo	0.81	0.57	0.64	0.81	0.57	0.64	0.81	0.57	0.64
Symphony	0.84	0.526	0.735	0.84	0.526	0.735	0.84	0.526	0.735
Blog	0.91	0.63	0.77	0.91	0.63	0.77	0.91	0.63	0.77
Latke	0.581	0.374	0.453	0.581	0.374	0.453	0.581	0.374	0.453
SSM-master	0.83	0.77	0.62	0.83	0.77	0.62	0.83	0.77	0.62
JHotDraw	0.85	0.71	0.75	0.85	0.71	0.75	0.85	0.71	0.75
JFreeChart	0.52	0.48	0.49	0.52	0.48	0.49	0.52	0.48	0.49

表 4　开源项目违规情况分布

项目名称	跨层依赖违规	反向依赖违规	非面向接口实现违规
Spring-petclinic	×	×	×
MyBlog	×	×	×
Junit3	×	×	√
Solo	√	×	√
Symphony	√	√	√
Blog	×	×	×
Latke	×	×	√
SSM-master	×	×	×
JHotDraw	×	×	√
JFreeChart	×	×	√

4.3　验证目的三:重构建议实施效果

以开源软件系统 Solo 为例,在重构点定位之后,用 LARTT 工具生成 207 条相应的代码重构建议,其中针对跨层违规切除依赖边的建议 12 条,针对非面向接口实现违规的代码重构建议 195 条. 本实验挑选其中 12 条针对跨层违规以及 20 条针对非面向接口实现违规的建议,进行重构建议实施. 实施前后 Solo 系统的合规度量指标变化如表 5 所示.

表 5　重构建议实施效果

	SCV	BCV	NIOVL
重构之前	0.6	1.0	0.3
重构之后	1.0	1.0	0.41
提升率	66.7%	0	36.7%

在对部分重构建议进行实施过后,Solo 系统的分层模式使用合规性得到了提升,关于跨层依赖的合规性提升到了最高值 1.0,提升了 66.7%;关于非面向接口实现违规的合规性提升了 36.7%. 考虑到只选择了部分重构建议进行实施,这样的提升效果可以证明 LARTT 生成的重构建议在一定程度上可以有效帮助重构人员进行重构实施,改善系统违规情况提升软件系统的质量.

5　结束语

本文提出基于分层架构模式识别的软件架构重构技术,通过使用代码词汇信息进行软件职责恢复、利用类结构图生成组件结构依赖图的方法进行软件架构模式识别,并以模式识别结果为基础,进行重构点定位和重构建议生成,最后通过实验分析验证本文该技术在模式识别有效性、重构点识别效果和重构建议实施效果方面比传统方法有较大提升,能够有效的帮助软件开发人员进行软件架构模式识别、定位重构点和生成重构建议,协助开发人员实施架构重构,改善系统违规情况,提升软件质量.

参考文献

[1] Silva L D, Balasubramaniam D. Controlling software architecture erosion: A survey[J]. Journal of Systems and Software, 2012, 85(1): 132 – 151.

[2] 杜鹏程. 基于软件职责和结构的分层架构模式识别及重构[D]. 江苏南京:东南大学, 2019.

[3] Clements P, Bachmann F, Bass L, et al. Documenting Software Architectures: Views and Beyond[M]. USA, 2010. 740 – 741.

[4] Chavan P U, Murugan M, Chavan P P. A review on software architecture styles with layered robotic software architecture[A]. International Conference on Computing Communication Control and Automation [C]. Pune, India, 2015. 827 – 831.

[5] Samarthyam G, Suryanarayana G, Sharma T. Refactoring for software architecture smells[A]. The 1st International Workshop[C]. USA: ACM, 2016. 1 – 4.

[6] Linhui Zhong, Haitao Ye, Jing Xia. Research on software evolution reconstruction based on architecture recovery[A]. IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)[C]. USA: IEEE, 2018. 68 – 71.

[7] Kong X, Li B, Wang L, et al. Directory-based dependency processing for software architecture recovery[J]. IEEE Access, 2018, 6: 52321 – 52335.

[8] Andreopoulos B, An A, Tzerpos V, Wang X. Clustering large software systems at multiple layers[J]. Information and Software Technology, 2007, 49(3): 244 – 254.

[9] Laval J, Anquetil N, Bhatti U, et al. OZone: Layer identification in the presence of cyclic dependencies[J]. Science of Computer Programming, 2013, 78(8): 1055 – 1072.

[10] 任武. 通过用况聚类促进软件结构恢复的方法[J]. 电子学报, 2013, 41(7): 166 – 172.
REN Wu. Approach for facilitating structural recovery by clustering use cases[J]. Acta Electronica Sinica, 2013, 41(7): 166 – 172. (in Chinese)

[11] 王 桐, 廖力, 李必信. 一种基于演进原则度量的软件架构持续演进效果评估方法[J]. 电子学报, 2019, 45(7): 1475 – 1481.
WANG Tong, LIAO Li, LI Bi-xin. An approach to evaluate the sustainable evolution effect of software architecture based on the measurements of evolution principles[J]. Acta Electronica Sinica, 2019, 45(7): 1475 – 1481. (in Chinese)

[12] 刘辉辉, 李必信, 廖力, 等. Java 类和包的易替换性度量

与影响因素分析[J]. 电子学报,2017,45(9):2149-2155.
LIU Hui-hui, LI Bi-xin, LIAO Li, et al. Replaceability measurement and impact factor analysis for java classes and packages[J]. Acta Electronica Sinica, 2017, 45(9): 2149-2155. (in Chinese)

[13] Binkley D, Heinz D, Lawrie D, et al. Understanding LDA in source code analysis[A]. International Conference on Program Comprehension[C]. USA: ACM, 2014. 26-36.

[14] 孙小兵,刘湘月,李斌,张伟佳. 基于相关主题模型的程序网络自动构建与分析[J]. 电子学报,2017,45(5):1052-1056.
SUN Xiao-bing, LIU Xiang yue, LI Bin, ZHANG Wei-jia. On automatic construction and analysis of program network via relational topic mode[J]. Acta Electronica Sinica, 2017, 45(5): 1052-1056. (in Chinese)

[15] Belle A B, Boussaidi G E, et al. Combining lexical and structural information to reconstruct software layers[J]. Information and Software Technology, 2016, 74(1): 1-16.

[16] Belle A B, El Boussaidi G, et al. The layered architecture revisited: is it an optimization problem[A]. The 25th International Conference on Software Engineering and Knowledge Engineering[C]. USA: SEKE, 2013. 1-6.

[17] Schmidt F, Macdonell S G, et al. An automatic architecture reconstruction and refactoring framework[A]. The 9th International Conference on Software Engineering Research, Management and Applications[C]. USA: SERA, 2011. 95-111.

[18] Elish K O, Alshayeb M. A classification of refactoring methods based on software quality attributes[J]. Arabian Journal for Science & Engineering, 2011, 36(7): 1253-1267.

[19] Sarkar S, Maskeri G, Ramachandran S. Discovery of architectural layers and measurement of layering violations in source code[J]. Journal of Systems and Software, 2009, 82(11): 1891-1905.

作者简介

王 丽 女,1982 年出生,江苏连云港人. 2009 年毕业于同济大学. 现为东南大学计算机科学与工程学院工程博士生,江苏自动化研究所高级工程师. 主要从事软件工程、软件质量与可靠性、软件测评等方面的研究工作.
E-mail: 230188058@seu.edu.cn

杜鹏程 男,1994 年出生,江苏东台人. 现为东南大学软件学院软件工程专业硕士研究生,研究方向为软件架构模式识别及重构.
E-mail: 15606135595@163.com

许一鸣 男,1993 年出生,江苏徐州人. 现为江苏自动化研究所工程师. 主要从事软件工程、软件质量与可靠性、软件测评等方面的研究工作.
E-mail: yimingxu@cumt.edu.cn

李必信(通讯作者) 男,1969 年出生,安徽人. 现为东南大学计算机科学与工程学院教授、博士生导师. 主要从事软件建模、分析、测试与验证、智能软件架构理论和方法、软件演化和软件质量保证等方面的研究工作.
E-mail: bx.li@seu.edu.cn