代码缺陷与代码味道的自动探测与优化研究*

刘 伟[1], 刘宏韬[2], 胡志刚[1,2]
(1. 中南大学 信息科学与工程学院, 长沙 410083; 2. 中南大学 软件学院, 长沙 410075)

摘 要: 为了实现代码缺陷与代码味道的自动探测与优化,提升优化与重构的效率,设计并开发了一套名为 SCORT 的源代码优化与重构工具。SCORT 将源代码解析为抽象语法树,再探测其中存在的代码缺陷和代码味道,最后对缺陷和味道进行自动优化和重构。在 SCORT 中已经实现了对 15 种常见代码缺陷和六种常见代码味道的检测以及自动优化与重构,提供了多种代码味道的自动重构算法,且具有良好的可扩展性。通过对三个待测项目的探测和优化实验结果表明,对于常见代码缺陷的探测和优化,SCORT 的精确率、召回率和准确率均可达 100%;对于部分常见的代码味道,SCORT 尚需进一步完善。SCORT 有助于开发人员提高代码质量,减少源代码中存在的缺陷和味道。

关键词: 代码缺陷; 代码味道; 探测; 优化; 重构; 抽象语法树
中图分类号: TP311.5 **文献标志码**: A **文章编号**: 1001-3695(2014)01-0170-07
doi:10.3969/j.issn.1001-3695.2014.01.040

Automated detection and optimization research on code defects and code smells

LIU Wei[1], LIU Hong-tao[2], HU Zhi-gang[1,2]
(1. *School of Information Science & Engineering, Central South University, Changsha* 410083, *China*; 2. *School of Software, Central South University, Changsha* 410075, *China*)

Abstract: To implement automated detection and optimization of code defects and code smells, and to improve the efficiency of code optimization and refactoring, this paper designed and developed a source code optimization and refactoring tool called SCORT. Firstly, SCORT parsed source code to an abstract syntax tree (AST), then detected the code defects and code smells in AST, finally, executed automated optimization and refactoring for existing defects and smells. SCORT could detect and optimize 15 kinds of common code defects and 6 kinds of common code smells. And it provided several code smells refactoring algorithms and had good scalability. Through detection and optimization experiments for three projects, the results show that for common code defects' detection and optimization, the precision, recall and accuracy can achieve 100% in SCORT, but for some common code smells, SCORT needs to be improved further. SCORT can help developers to improve code quality, and reduce defects and bad smells in source code.

Key words: code defects; code smells; detection; optimization; refactoring; abstract syntax tree

0 引言

软件开发效率和代码质量在软件开发过程中占据着非常重要的位置,不良的代码和低劣的设计会大幅降低开发效率和代码质量,并影响到软件的性能、健壮性和可维护性,同时增加软件开发和维护成本。因此设计并编写高质量的代码一直是软件开发所追求的目标之一,开发者往往会通过各种方法来保证软件代码的灵活性和可靠性,也极力促使代码更容易被阅读和理解。

由于程序本身的复杂性和软件开发人员的经验及能力等因素,在软件设计和程序代码中难免会存在一些问题。软件工程大师 Fowler 等人[1]使用代码味道(code smells)来描述那些差劲的设计和不良的编码习惯,代码味道实际上指的是代码坏味(bad smells in code)。为了提高软件质量,需要对软件设计和程序代码中的味道进行合理的优化与重构。重构(refactoring)是指在不改变外部行为的前提下,有条不紊地改善代码,它可以最大限度地减少整理过程中引入错误的几率。在文献[1]中,Fowler 等人定义了 22 种常见的代码味道,包括重复代码、发散式变化、霰弹式修改、数据泥团、基本类型偏执等。此后,又有研究人员陆续提出了一些新的代码味道,Kerievsky 在文献[2]中又提出了五种新的代码味道,包括条件逻辑太复杂、不恰当的暴露、解决方案蔓延、组合爆炸等。除了代码味道,许多程序源代码还存在一些缺陷,如未使用的参数或局部变量、空的条件分支语句、属性名和方法名重复等。代码缺陷和代码味道并没有严格的分界线,有些代码味道本身就是代码缺陷,如过大的类、过长的函数等。

探测和优化代码中的缺陷与味道对于提高代码质量具有非常重要的意义[3]。通常情况下,代码缺陷和代码味道的探测和优化由开发人员手工完成,效率较低,而且很大程度上依

收稿日期: 2013-05-05; **修回日期**: 2013-06-13 **基金项目**: 国家自然科学基金资助项目(60970038, 61272148);湖南省研究生科研创新基金资助项目(CX2012B068)

作者简介: 刘伟(1982-),男,湖南娄底人,高级工程师,博士研究生,主要研究方向为软件工程、数据挖掘(weiliu@csu.edu.cn);刘宏韬(1991-),男,湖南长沙人,硕士研究生,主要研究方向为软件工程;胡志刚(1963-),男,山西孝义人,教授,博导,博士,主要研究方向为软件工程、分布式计算.

赖于开发人员的经验,如果之前没有相关的经验可能会导致代码更加混乱。若能使用自动化工具探测软件产品中的代码缺陷和代码味道,并能对它们针对性地应用一些手段进行优化和重构,必然能够提高软件的可靠性、可维护性和可理解性,同时降低软件开发的成本,在提升代码质量的同时提高开发效率。

1　相关工作

近年来,代码缺陷和代码味道的自动探测与优化已成为软件工程研究的一个热点领域[4~8],诞生了一系列的方法和工具,部分研究成果已经在开发界得以应用。

现有的代码自动分析与优化工具倾向于从技术角度提升代码质量,代码段中的缺陷越少,质量也就越高。代码分析与优化归结为低级的缺陷追踪,而这些可以从一些查找指针算法、内存分配、空引用、数组边界错误等问题的工具反映出来,这类分析和优化工具又称为静态代码分析工具,包括 PMD[9]、CheckStyle[10]和 FindBugs[11]等。其中,PMD 可以检查 Java 源代码并寻找潜在的问题,探测诸如命名问题、不用的代码、空的语法模块、未使用的私有变量或参数等代码缺陷,也能找出一些代码味道,如过大的类、过长函数、过长参数列和重复代码,并且允许用户设定找出这些代码味道的阈值以实现度量手段,还能通过 Java 和 XPath 两种方法来添加探测的规则,以便更好地扩展;CheckStyle 能够帮助开发人员坚持以一种编码规范来编写 Java 代码,它探测代码大小是否违规、类的设计是否良好,以及变量声明和一些代码的问题,同时也能找出过大的类、过长函数、过长参数列、重复代码等代码味道;FindBugs 由马里兰大学 Pugh 教授领导的研究小组所开发,它可以检查 class 文件和 JAR 文件,将字节码与一组缺陷模式(bug patterns)进行比对来发现可能存在的问题。它与前两个工具最大的区别在于它的目标文件是字节码文件而不是程序源文件,能够检查的缺陷模式包括空指针的查找、是否存在未关闭的资源、字符串内容比较错误(如误用“==”来比较字符串内容是否相同)等。这三款开源工具都发布于全球最大的开源软件开发平台 SourceForge,它们暂时都不能探测和优化 switch 惊悚现身、冗余类等 Fowler 等人[1]提到的代码味道,同时也没有采用工具或脚本语言来描述如狎昵关系等复杂的代码味道。除了上述开源工具之外,还存在一些商业的代码优化与重构工具,如 inFusion[12]等。通常,这些商业工具能够探测到更多的代码缺陷和代码味道。

Fontana 等人[6]对一些常用的代码优化工具所能够探测的代码味道进行了总结,这些工具大部分都不能在探测到代码坏味之后能够对代码进行自动化重构。

此外,一些国际知名软件公司也开展了相关工具的研发工作,例如 IBM 公司推出的 checking tool for bugs errors and mistakes(BEAM)工具[13]、Parasoft 公司推出的代码分析和单元测试工具 Jtest[14]、Microsoft 公司也开展了包含代码分析与优化的研究项目 Phoenix[15]。同时,在一些主流 IDE(integrated development environment,集成开发环境)中也集成了一些简单的代码优化和重构功能,如 IBM Eclipse 和 Microsoft Visual Studio 中的重构(refactoring)菜单,但所提供的功能仅限于接口的提取、属性的封装、公共方法提取等最简单的重构技巧。例如,在 Visual Studio 中,包含了重命名、提取代码为新方法、提取私有变量为属性等少数的重构手法;而与之相比,Eclipse 中所提供的重构手段则更多一些,除了提取方法、封装域等简单的重构外,还有引入工厂等较为复杂的重构。但集成在 IDE 中的优化与重构功能很有限,功能的完善程度和重构的自动化程度都有待提高。

目前国内从事代码味道探测和重构研究的学者和研究机构并不多,北京理工大学 Liu 等人[7,8]近年在该领域取得了不少成果,发表了多篇高水平学术论文,他们针对各种不同类型的代码味道,分析了不同代码味道之间的关系,绘制了一张代码味道探测与解析序列图,从而简化了检测与解析的难度,提高了味道探测的效率[7];此外,他们还研究了如何识别泛化重构机会,通过概念关系、实现相似性、结构一致性和继承体系结构分析等来识别潜在的重构机会,并研制了一款名为 GenReferee 的工具来自动推荐泛化重构机会[8]。

已有的绝大多数代码分析与优化工具如 PMD、CheckStyle、FindBugs、GenReferee 等,虽然能够有效地识别部分代码缺陷和味道,但是它们不能实施代码的自动化重构[6]。然而在很多时候,用户希望代码检测工具能够引导他们如何对代码缺陷和味道进行优化和重构,最好能够提供自动重构功能,并对重构之前的代码和重构之后的代码进行可视化对比,让他们避免以后再出现类似的问题。此外,代码的自动优化与重构可以提高优化和重构的效率,减少人工重构中引入的错误。因此,研究如何对代码缺陷和代码味道进行自动化重构,以及设计和开发自动优化与重构工具具有较强的理论价值和实用价值。

2　SCORT 的分析与设计

本文研究并开发了一套源代码自动优化与重构工具——SCORT(source code optimization and refactoring tool)。SCORT 通过对源代码生成的 AST(abstract syntax tree,抽象语法树)进行分析,实现了对常见代码缺陷和代码味道的自动探测,并能够对代码进行自动优化与重构。对于每一种代码缺陷,SCORT 充分考虑到多种变化形式,能够探测出同一代码缺陷的不同变形,并能实现对代码的自动优化;对于每一种代码味道,SCORT 提供了一套完整的自动重构算法,在保证功能正确的前提下提高代码的质量。

2.1　Eclipse AST

SCORT 在对 Java 源代码进行分析时,使用了 Eclipse 提供的 JDT(Java development tools,Java 开发工具包)和 Eclipse AST。Eclipse JDT 提供了一组访问和操作 Java 源代码的 API(application programming interface,应用程序接口),它允许开发人员访问工作空间中已经存在的项目、创建新项目、修改已存在的项目,它还允许用户启动 Java 程序。JDT 提供了两种不同的方式来访问 Java 源代码,即 Java 模型和抽象语法树。

SCORT 的设计与实现基于 Eclipse AST[16]。Eclipse AST 是 Eclipse JDT 的一个重要组成部分,它定义在 Eclipse 插件 org. eclipse. jdt. core 的包 org. eclipse. jdt. core. dom 中,AST 定义了用于修改、创建、读取和删除源代码的 API。Eclipse AST 采用工厂方法模式和访问者模式来设计和实现,可以减轻用户深入了解其内部结构的压力,并且方便用户利用它们构建并处理 AST。

Eclipse AST 提供了一个 ASTNode 类,Java 源代码中的每一个语法结构都对应 ASTNode 的一个子类,大多数节点类的名称和意义都很明确,如 Comment(注释)、PackageDeclaration

(包声明)、MethodDeclaration(方法声明)、VariableDeclarationFragment(变量声明)、SimpleName(非 Java 关键字字符串)等,Java 类用 Compilation unit(编译单元)节点表示。Eclipse AST 提供一个用于创建 AST 节点的工厂类 AST,该类中包含许多创建各类 AST 节点的工厂方法,用户可以利用这些方法来构建一棵 AST。此外,Eclipse AST 还提供了一个 ASTVisitor 类用于定义 AST 的抽象访问者,在该类中声明了一组访问各类 AST 节点的方法,包括 visit()、endVisit()和 preVisit()方法,这些方法都通过参数接收一个 AST 节点类型的对象,然后对该节点进行访问以便执行一些操作。

2.2 系统整体设计

在 SCORT 中,使用抽象语法树来表示 Java 源代码。首先使用 Eclipse AST 读入 Java 源代码,然后将其解析为抽象语法树,该抽象语法树包含代码所有结构信息,通过使用访问者模式定义不同的具体访问类来访问抽象语法树中的节点,探测出代码中的缺陷和味道,再对这些缺陷和味道进行优化与重构。SCORT 的整体流程如图 1 所示。

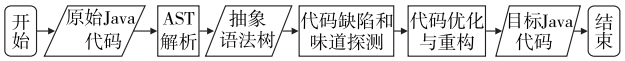

图 1 SCORT 整体流程

在 SCORT 的具体实现中,将代码缺陷和味道探测以及代码优化与重构两个步骤融为一体。针对代码中普遍存在的缺陷和味道,定义了一系列 XXXDetector 类,每一个 XXXDetector 类负责探测并处理一种缺陷或味道。为了让系统具备更好的灵活性和可扩展性,结合职责链模式[17]来设计 SCORT,如图 2 所示。

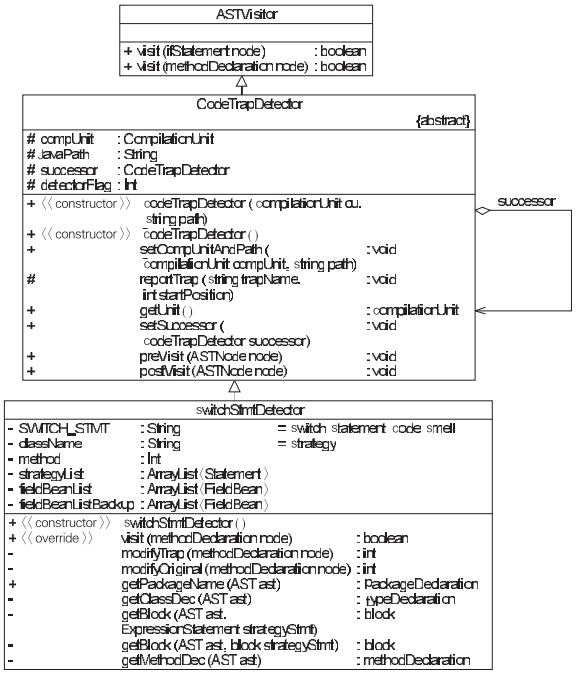

图 2 SCORT 类图片段

在图 2 中,提供了一个抽象类 CodeTrapDetector 作为所有 Detector 类的公共父类,充当职责链模式中的抽象处理者,每一个具体的 Detector 类作为具体处理者,CodeTrapDetector 类包含一个自关联,用于设置后续的 Detector 类,多个 Detector 类可以连成一条处理链,从而实现对多个缺陷和味道的探测与优化。为了让链的创建更加灵活,引入了一个名为 ChainConfig. xml 的 XML 格式的配置文件,代码片段如下:

```
〈?xml version = "1.0"?〉
〈config〉
    〈codetrap〉EmptyIfStmtDetector 〈/codetrap〉
    〈codetrap〉LongParamListDetector 〈/codetrap〉
    〈codetrap〉SwitchStmtDetector 〈/codetrap〉
〈/config〉
```

在 SCORT 中,提供一个工具类 chainXMLUtil 用于读取配置文件,根据配置文件中存储的具体处理类的类名创建相应的对象,并创建一条处理链,实现多种缺陷和味道的批量处理。

SCORT 的界面如图 3 所示,它以 Eclipse 插件的形式提供给用户使用。SCORT 通过对源代码进行分析并找出其中存在的代码缺陷和代码味道,再通过代码对比界面来呈现重构之前和重构之后的代码差异,用户可以通过 SCORT 对代码进行自动优化和重构。

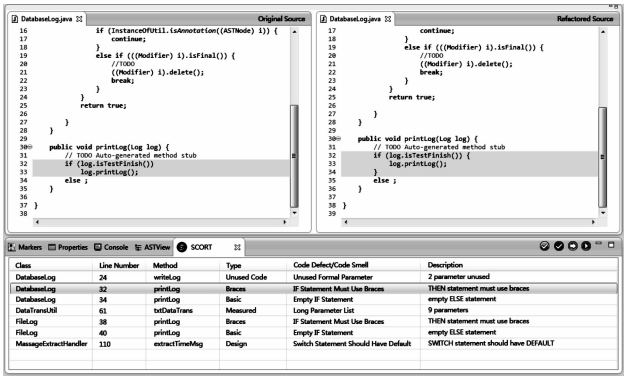

图 3 SCORT 界面截图

3 代码缺陷检测与自动优化

SCORT 已实现 15 种代码缺陷的检测与自动优化,这些代码缺陷包括基本、括号、设计、命名和未使用的代码等类别,如表 1 所示。下面选取代码缺陷“空的 if 语句块”(empty if statement)和“未使用的参数”(unused formal parameter)加以详细说明。

表 1 SCORT 已实现代码缺陷一览表

分类	名称	说明
基本(basic)	empty if statement	空的 if 语句块
	empty switch statements	空的 switch 语句块
	empty try block	空的 try 语句块
	empty while statement	空的 while 语句块
括号(braces)	for loops must use braces	for 循环未使用大括号
	if statement must use braces	if 语句块未使用大括号
	while loops must use braces	while 未使用大括号
设计(design)	avoid final local variable	避免使用“final”局部变量
	switch statement should have default	switch 语句块没有缺省语句
命名(naming)	avoid field name matching method name	避免变量名和方法名一致
	avoid field name matching type name	避免变量名和类名一致
未使用的代码(unused code)	unused formal parameter	未使用的参数
	unused local variable	未使用的局部变量
	unused private field	未使用的私有类变量
	unused private method	未使用的私有方法

3.1 空的 if 语句块

空的 if 语句块是一种很常见的代码缺陷。由于软件开发人员的人为失误以及一些代码生成工具只能生成程序框架代码等原因,导致代码中可能会存在一些空的 if 语句块,这些无用的语句将影响到代码的编译与执行效率,也降低了代码的可

读性和易理解性,因此,需要自动去除空的 if 语句块。在 SCORT 中,通过扩展 CodeTrapDetector 类创建一个名为 EmptyIfStmtDetector 的子类来探测并优化空的 if 语句块。

通过分析,找出了九种典型的空的 if 语句块的代码模式,针对不同的代码模式使用不同的优化方法,这九种代码模式及优化方式说明如表 2 所示。

表 2　空的 if 语句块代码模式及优化方式

类型	编号	代码模式	描述	优化方式
then 语句	1	if(condition); then 语句块中无语句	无 then 语句块,仅有一空语句	删除 if 语句块
	2	if(condition){}		
	3	if(condition){;}	then 语句块中仅有一空语句	将 else 语句转换为 then 语句,并将 if 条件取反,即重构为 if(!(condition)){statement}的形式
	4	if(condition); else {statement}	无 then 语句块,但 else 语句块不为空	
	5	if(condition){} else {statement}	then 语句块中无语句,但 else 语句块不为空	
	6	if(condition){;} else {statement}	then 语句块中仅有一空语句,但 else 语句块不为空	
else 语句	7	if(condition) {statement} else;	无 else 语句块,仅有一空语句	删除 else 语句块
	8	if(condition) {statement} else {}	else 语句块中无语句	
	9	if(condition) {statement} else {;}	else 语句块中仅有一空语句	

图 4 是一个空的 if 语句块优化实例,对应于表 2 中的代码模式 7,当 then 语句块中无语句,且 else 语句块不为空时,SCORT 自动将 else 语句转换为 then 语句,并将 if 条件取反。

```
……
if (flag! =1 || numOfActivity ==0){
    count ++;
    numOfActivity ++;
    if (flag >=2){
    }
    else {
        --count;
    }
}
else;
……
```

⇨

```
……
if (flag! =1 || numOfActivity ==0){
    count ++;
    numOfActivity ++;
    if (flag >=2){
        --count;
    }
}
……
```

图 4　空的 if 语句块优化实例

3.2　未使用的参数

在源代码中,有时候由于软件开发人员的疏忽或者考虑到未来扩展的方便性(实质上有些考虑完全是多余的),导致方法中有些参数在方法体中从未被使用。这些参数的存在使得方法签名变得复杂,给方法的调用带来很多不便,客户端代码需要给这些未使用的参数设置实参,而这些参数无论取什么值实质上都没有任何意义。此时,需要删除这些从未被使用的参数。在 SCORT 中,通过扩展 CodeTrapDetector 类创建一个子类 UnusedFormalParamDetector 来探测并优化这些未使用的参数。

在 UnusedFormalParamDetector 类中,首先获取一个方法的参数列表,然后将参数列表中参数的使用情况委派给 UnusedParamVisitor 类来处理,UnusedParamVisitor 类是 ASTVisitor 的一个子类,覆盖了 visit(SimpleName node)方法。先假定参数列表中的每一个参数都是未被使用的参数,存储在未使用参数列表中,当方法体中的每一个非 Java 关键字字符串出现时,将执行自定义的 visit(SimpleName node)方法,如果该关键字字符串与参数列表中的某个参数名称相等,说明该参数被使用过,则从未使用参数列表中删除该参数。待整个方法体全部处理完毕后,再检查未使用参数的列表,如果不为空,则调用相关方法将原方法中遗留的参数删除,这些参数就是从未被使用过的参数。UnusedFormalParamDetector 类与 UnusedParamVisitor 类之间的关系和它们的常用方法如图 5 所示。

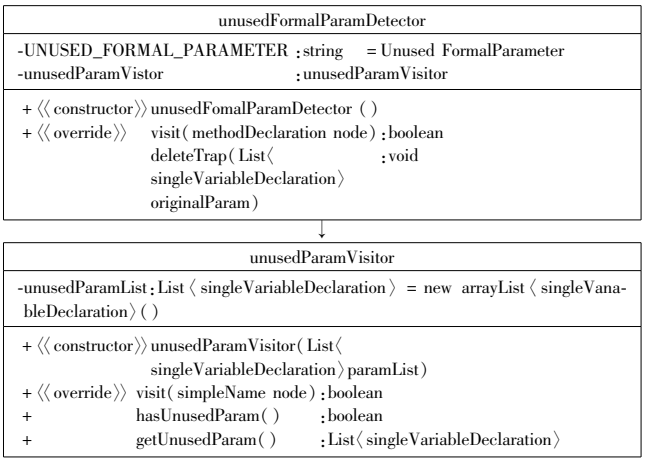

图 5　UnusedFormalParamDetector 和 UnusedParamVisitor 的类图

图 6 是一个未使用的参数优化实例,通过检测和优化,SCORT 会自动将 method()方法中多余的参数 unusedBool 和 unusedParamA 去除。

```
……
public void method(int flag,boolean unusedBool,string name,arrayList〈Demo[]〉unusedParamA,int count,int numOfActivity){
    count ++;
    operation(flag,name,numOfActivity);
}
……
```

⇩

```
……
public void method(int flag,string name,int count,int numOfActivity){
    count ++;
    operation(flag,name,numOfActivity);
}
……
```

图 6　未使用的参数优化实例

4　代码味道探测与自动重构

SCORT 已实现六种 Fowler 等人所定义的代码味道的检测与自动优化,这些代码味道包括可度量、重复和条件逻辑等类型,如表 3 所示。

表 3　SCORT 已实现代码味道一览表

分类	名称	简要说明
可度量(measured)	过长函数(long method)	函数(方法)太大,代码冗长,维护和复用困难
	过大的类(large class)	类太大,职责太重,扩展和复用困难
	过长参数列(long parameter list)	方法参数列太长,难以理解,容易造成不一致
重复(duplication)	重复代码(duplicated code)	不同的地方出现相同的程序代码和程序结构
	异曲同工的类(alternative classes with different interfaces)	做相同事情的方法(函数)却具有不同的方法签名
条件逻辑(conditional logic)	switch 惊悚现身(switch statements)	代码中存在冗长的条件判断语句

下面选取代码味道过长参数列(long parameter list)和 switch 惊悚现身(switch statements)加以详细说明。

4.1　过长参数列

如果一个方法拥有过长的参数列,将导致参数列表难以理

解,而且太多参数容易造成前后不一致、不易使用,客户端在调用时也较为麻烦,需要仔细对照参数的个数、类型和顺序,以保证形参与实参的一致性。因此过长的参数列是代码中的一种坏味道,Fowler 等人针对该味道提出了一些重构方案,如 replace parameter with method(以函数取代参数)、preserve whole object(保持对象完整)和 introduce parameter object(引入参数对象)等[1]。其中,引入参数对象是最常用的解决方法,它也是处理另一种代码味道 data clumps(数据泥团)时常用的一种重构方法。

在 SCORT 中,通过扩展 CodeTrapDetector 类创建一个子类 LongParamListDetector 来探测并重构过长参数列。将方法中的参数封装在一个对象中,为了保持数据的封装性,所有的参数在封装成属性后,它们的可见性都设置为私有(private),并为每一个参数所对应的属性提供一对 getter 和 setter 函数来向外界暴露其内容。因此,需要对参数的各种使用情况进行综合考虑,通过分析,找出了 12 种参数使用模式,如表 4 所示,在使用参数对象进行代码重构时将逐一考虑每一种代码模式,为不同的代码模式提供了不同的优化方式。SCORT 中参数的临界数值(具有多少个参数时进行自动重构)由用户来决定,通过修改配置文件可改变参数的临界数值。

表 4 过长参数列模式及优化方式一览表

类型	编号	代码模式	描述	优化方式
赋值(assignment)	1	paramName = value;	父节点为表达式语句	paramObj. setParamName(value)
	2	(paramName = value)	父节点为带括号的表达式	(paramObj. setParamName(value))
	3	paramName operator = value	赋值操作符为快捷赋值操作	paramObj. setParamName(paramObj. getParamName() operator value)
函数调用(method invocation)	4	method(paramName)	作为参数存在于函数调用中	method(paramObj. getParamName())
前缀表达式(prefix expression)	5	! paramName	否定前缀表达式(参数为 boolean 类型)	! paramObj. isParamName()
	6	++/--paramName;	自增或自减的前缀形式,且父节点为表达式语句	paramObj. setParamName(paramObj. getParamName() +/- 1);
	7	value operator ++/--paramName	自增或自减的前缀形式,并且父节点为中缀表达式	paramObj. setParamName(paramObj. getParamName() +/-1); value operator paramObj. getParamName()
中缀表达式(infix expression)	8	paramName operator value	为中缀表达式的左操作数	paramObj. getParam. Name() operator value
	9	value operator paramName	为中缀表达式的右操作数	value operator paramObj. getParamName()
后缀表达式(postfix expression)	10	paramName ++/--;	自增或自减的后缀形式,且父节点为表达式语句	paramObj. setParamName(paramObj. getParamName() +/- 1);
	11	value operator paramName ++/--	自增或自减的后缀形式,且父节点为中缀表达式	value operator paramObj. getParamName() paramObj. setParamName(paramObj. getParamName() +/- 1);
数组(array access)	12	paramName[i]	以数组形式出现	paramObj. getParam Name()[i]

过长参数列的具体重构流程如图 7 所示。

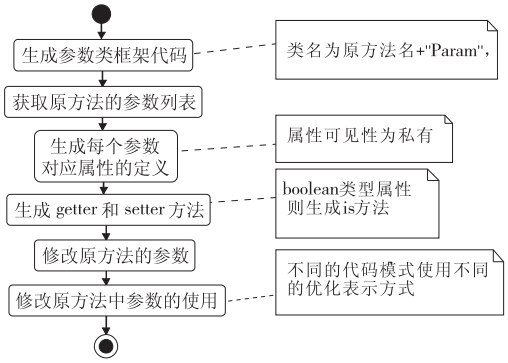

图 7 过长参数列重构流程

图 8 是一个过长参数列重构实例,在重构之前,方法 method()包含三个参数,通过引入参数类对参数列进行重构,SCORT 将自动创建一个名为 MethodParam 的参数类,该类封装了每一个参数的 getter 和 setter 方法。

在如图 8 所示的 method()方法中,原有的三个参数都被封装在 MethodParam 类型的对象中,方法参数也被重构为只有一个 MethodParam 类型的参数对象 param。由于 SCORT 在对过长参数列进行重构时,参数类中所有属性的可见性都是私有的,必须通过 getter 和 setter 函数来访问,因此导致重构之后参数的使用会比重构之前复杂,除非将参数类中所有属性的可见性定义为公开的,否则这种复杂性的增加将无法避免。

```
……
public void method(int flag,int count,boolean numOfActivity){
    if(1 == flag + count || numOfActivity){
        count += flag;
    }
    else {
        count ++;
        operation(flag,numOfActivity);
    }
}
……
```

⇩

```
……
public void method(methodParam param){
    if(1 == param. getFlag() + param. getCount() || param. isNumOfActivity
()){
        param. setCount(param. getCount() + param. getFlag());
    }
    else {
    param. setCount(param. getCount() + 1);
        operation(param. getFlag(),param. isNumOfActivity());}
}
……
```

图 8 过长参数列重构实例

4.2 Switch 惊悚现身

在面向对象编程中提倡尽量少用甚至不用复杂的条件语句,包括 if 语句和 switch 语句。源代码中大量冗长的条件语句的存在,既不利于程序的测试和维护,还将导致系统难以扩展,增加一个新的分支不得不修改源代码,违反了面向对象设计的基本原则——开闭原则。除此之外,代码的复用性也将受到影响,难以实现只复用多重条件中某一个分支中的语句。因此,探测源代码中的条件语句并对其进行重构对改善代码质量而言意义重大。Fowler 等人将这种条件语句称为 switch 惊悚现身(switch statements),这是一种很常见的代码坏味道[1]。

在文献[1]中提到了多种对 switch 惊悚现身进行重构的方法,很多方法都需要结合多态来实现,即 replace conditional

with polymorphism(以多态取代条件式)。在很多情况下,switch-case 或 if-then-else 语句都通过检查类型码(type code)并根据不同的类型码的值执行不同的动作。最常用的一种重构方法是针对每种 type code 都建立相应的子类,即使用 replace type code with subclasses(以子类取代类型码)这一重构手段。具体流程如图 9 所示。

图 10 是一个 switch 惊悚现身重构实例,在重构之前,ShapeDrawing 类可以绘制各种不同类型的图形,在 draw( )方法中,通过判断类型码 shape_type 来绘制相应的图形,各种类型图形的绘制都由 draw( )方法来实现,导致代码存在过长函数、过大类、switch 惊悚现身等坏味道。在 SCORT 中,可以按照图 9 所示流程实现代码的自动重构,将不同图形的绘制程序封装到 ShapeDrawing 类的子类中,为不同类型的图形提供专门的绘图类,增加新的图形只需增加新的绘图子类即可,系统具有更好的可扩展性。

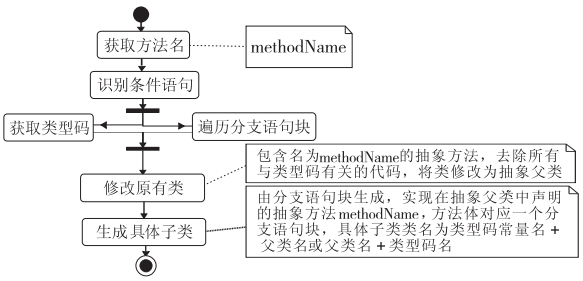

图 9　Switch 惊悚现身重构流程

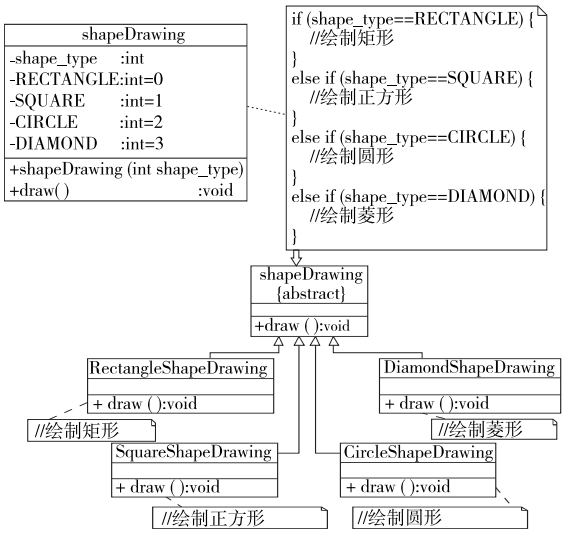

图 10　Switch 惊悚现身重构实例

5　实验结果及分析

为了验证 SCORT 的正确性,本文针对每一种代码缺陷和代码味道都构造了一系列测试用例,所有测试用例均由未参与 SCORT 设计与开发的人员提供,部分测试用例基于对一些 Java 开源软件的修改。实验结果使用精确率(precision)、召回率(recall)和准确率(accuracy)来评价。为了计算精确率、召回率和准确率,定义了以下四个值:a)TP(true positive)表示本应该优化且得到正确优化的测试用例数量;b)FP(false positive)表示本不应该优化却得到优化的测试用例数量;c)TN(true negative)表示本不应该优化最后也未被优化的测试用例数量;d)FN(false negative)表示本应该优化但未被优化的测试用例数量。

精确率、召回率和准确率的定义如下:

$$\text{精确率} = \frac{TP}{TP + FP},$$

$$\text{召回率} = \frac{TP}{TP + FN}$$

$$\text{准确率} = \frac{TP + TN}{TP + FP + TN + FN}$$

针对 SCORT 当前版本已经实现的 15 种代码缺陷和六种代码味道的自动探测与优化,由三名未参与 SCORT 设计与开发的人员通过分别修改三个 Java 项目进行测试,这三个项目的基本信息如表 5 所示。

表 5　待测项目基本信息一览表

项目名称	版本	类(接口)的数量	代码行数
ChatRoom	0.91	62	3 236
JHotDraw	5.3	249	14 611
Log4J	1.2.1	203	10 224

通过人工修改,在三个待测项目中引入一些测试用例。具体而言,针对每种代码缺陷设计 20 个测试用例;针对每种代码味道设计 10 个测试用例。具体分布情况如表 6 所示。

表 6　待测项目中代码缺陷和代码味道分布情况

代码缺陷/味道名称	ChatRoom	JHotDraw	Log4J
空的 if 语句块	8	6	6
空的 switch 语句块	8	6	6
空的 try 语句块	8	6	6
空的 while 语句块	8	6	6
for 循环未使用大括号	6	7	7
if 语句块未使用大括号	6	7	7
while 未使用大括号	6	7	7
避免使用"final"局部变量	10	6	4
switch 语句块没有缺省语句	10	6	4
避免变量名和方法名一致	8	6	6
避免变量名和类名一致	8	6	6
未使用的参数	10	6	4
未使用的局部变量	10	6	4
未使用的私有类变量	10	6	4
未使用的私有方法	10	6	4
过长函数(代码行数≥40)	4	4	2
过大的类(方法个数≥10)	4	4	2
过长参数列(参数个数≥5)	6	2	2
重复代码(重复代码行数≥5)	4	3	3
异曲同工的类	6	2	2
switch 惊悚现身	4	4	2

通过人工分析优化前和优化后的代码,待测项目中存在的代码缺陷的评估结果如表 7 所示。

表 7　SCORT 中代码缺陷探测与优化的精确率、召回率和准确率评估结果

代码缺陷名称	精确率/%	召回率/%	准确率/%
空的 if 语句块	100	100	100
空的 switch 语句块	100	100	100
空的 try 语句块	100	100	100
空的 while 语句块	100	100	100
for 循环未使用大括号	100	100	100
if 语句块未使用大括号	100	100	100
while 未使用大括号	100	100	100
避免使用"final"局部变量	100	100	100
switch 语句块没有缺省语句	100	100	100
避免变量名和方法名一致	100	100	100
避免变量名和类名一致	100	100	100
未使用的参数	100	100	100
未使用的局部变量	100	100	100
未使用的私有类变量	100	100	100
未使用的私有方法	100	100	100

由表 7 可知,对于常见的代码缺陷的探测和优化,SCORT 的精确率、召回率和准确率都达到了 100% 。

待测项目中存在的代码味道的评估结果如表 8 所示。由于原有代码中本身存在大量的过长函数与过长类,导致结果的分析过程非常复杂。表 8 只针对新引入的 10 个测试用例进行分析和评估。

表 8 SCORT 中代码味道探测与优化的精确率、召回率和准确率评估结果

代码味道名称	精确率/%	召回率/%	准确率/%
过长函数(代码行数≥40)	77.78	100	80
过大的类(方法个数≥10)	66.67	100	70
过长参数列(参数个数≥5)	100	100	100
重复代码(重复代码行数≥5)	83.33	62.5	60
异曲同工的类	100	62.5	70
switch 惊悚现身	80	50	50

由表 8 可知,对于代码味道的探测与优化的精确率、召回率和准确率,SCORT 尚有提高和改进的空间,分析其原因如下:

a)自动重构与人工重构的时机有所差异。例如,有些过长函数是由于大量的局部变量声明造成,如果是人工重构通常不会将这些变量声明代码单独提取为独立的函数;有些过大的类是因为包含大量的 getter 和 setter 函数,此时也无须进行重构,没有必要将这个类拆分为多个类。

b)在重复代码识别和异曲同工的类中,暂时只考虑完全相同的代码,尚未考虑参数化的重复代码以及等价代码。例如没有考虑两段只有局部变量名不同的代码或者两段等价的循环代码,导致 FN 值较大,需要在今后进一步改进。

c)在 switch 惊悚现身的探测与优化中,暂时只考虑单条件的判定语句,尚未考虑多条件的判定语句以及嵌套的判定语句,该味道的探测与优化功能有待进一步完善。

6 结束语

代码缺陷与代码味道的自动探测与优化对于提高软件质量具有重要意义。在本文中,研究并开发了一套源代码自动优化与重构工具 SCORT。SCORT 首先将源代码解析为抽象语法树,然后对存在的代码缺陷和代码味道进行探测,最后对这些代码缺陷和代码味道进行自动优化和重构。

SCORT 已经实现对 15 种常见代码缺陷以及六种常见代码味道的检测与自动优化,针对每一种缺陷和味道都设计并实现了相应的探测和优化算法。SCORT 对代码缺陷的各种变形进行了充分考虑,能够探测出同一代码缺陷的不同变形,并能实现对代码的自动优化。对于一些复杂的代码味道,SCORT 提供了一套完整的自动探测与优化算法,在保证功能正确的前提下,提高代码的质量。本文分别选取了两种代码缺陷和代码味道进行了详细说明并提供了相应的优化与重构流程和实例。通过实验验证,SCORT 对常见的 15 种代码缺陷的探测和优化的精确率、召回率和准确率达到了 100%,且对常见的六种代码味道的检测和重构也具有较高的精确率、召回率和准确率。

在 SCORT 的设计与开发中,合理使用了一些设计模式,如职责链模式等,使得 SCORT 具有良好的可扩展性,易于增加新代码缺陷和代码味道的探测与优化方法。

在后续工作中,将对 SCORT 进行进一步完善,具体工作包括:

a)将进一步完善现有的代码味道探测与重构算法,提高代码味道探测与重构的精确率、召回率和准确率。

b)将实现对更多代码缺陷和代码味道的优化与重构,结合设计模式,实现一些以设计模式为导向的重构。

c)充分利用各种代码缺陷和代码味道之间的关系,避免一些重复的探测与优化工作,提高系统的性能。

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] KERIEVSKY J. Refactoring to patterns[M]. Massachusetts: Addison-Wesley, 2004.

[3] Van EMDEN E, MOONEN L. Java quality assurance by detecting code smells[C]//Proc of the 9th Working Conference on Reverse Engineering. 2002:97-106.

[4] MOHA N, GUEHENEUC Y G, DUCHIEN L, *et al*. DECOR: a method for the specification and detection of code and design smells [J]. IEEE Trans on Software Engineering, 2010, 36(1): 20-36.

[5] ZHANG Min, HALL T, BADDOO N. Code bad smells: a review of current knowledge[J]. Journal of Software Maintenance and Evolution: Research and Practice, 2011, 23(3): 179-202.

[6] FONTANA F A, BRAIONE P, ZANONI M. Automatic detection of bad smells in code: an experimental assessment[J]. Journal of Object Technology, 2012, 11(2): 1-38.

[7] LIU Hui, MA Zhi-yi, SHAO Wei-zhong, *et al*. Schedule of bad smell detection and resolution: a new way to save effort[J]. IEEE Trans on Software Engineering, 2012, 38(1): 220-235.

[8] LIU Hui, NIU Zhen-dong, MA Zhi-yi, *et al*. Identification of generalization refactoring opportunities[J]. Automated Software Engineering, 2013, 20(1): 81-110.

[9] DANGEL A, PELISSE R. PMD[CP/OL]. (2013-05-01)[2013-05-04]. http://sourceforge.net/projects/pmd/.

[10] BURN O. CheckStyle [CP/OL]. (2012-10-12)[2013-05-04]. http://sourceforge.net/projects/checkstyle/.

[11] HOVEMEYER D, PUGH B. FindBugs [CP/OL]. (2013-04-08)[2013-05-04]. http://sourceforge.net/projects/findbugs/.

[12] INTOOITUS S R L. inFusion[CP/OL]. (2012-10-31)[2013-05-04]. http://www.intooitus.com/products/ infusion.

[13] BRAND D. Error detection by data flow analysis restricted to executable paths[EB/OL]. (1999-05-24)[2013-05-04]. http://www.research.ibm.com/da/publications/beam_data_flow.pdf.

[14] Parasoft. Jtest[CP/OL]. (2013-04-22)[2013-05-04]. http://www.parasoft.com/jsp/products/jtest.jsp.

[15] Microsoft Research. Phoenix compiler and shared source common language infrastructure[EB/OL]. (2013-04-22)[2013-05-04]. http://research.microsoft.com/en-us/ collaboration/focus/cs/phoenix.aspx.

[16] THOMAS K, EYE M G, OLIVIER T. Eclipse corner article: abstract syntax tree[EB/OL]. (2006-11-20)[2013-05-04]. http:// www.eclipse.org/articles/article.php? file = Article-JavaCodeManipulation_AST/index.html.

[17] GAMMA E, HELM R, JOHNSON R, *et al*. Design patterns: elements of reusable object-oriented software[M]. Massachusetts: Addison-Wesley, 1995.