面向对象软件系统中去除 code smell 的重构技术综述

摘要：在日常软件开发过程中，由于软件开发人员的水平参差不齐或者系统架构设计上的不当很容易引入 code smell，降低软件质量并且不利于后续的软件开发工作甚至可能增加系统发生故障或错误的风险。因此 code smell 的检测在软件维护过程中是非常必要的。重构是一种在不更改软件外部行为的情况下改变软件内部结构的活动，通过重构去除 code smell 能够显著改善软件质量。本文将介绍若干面向对象系统中去除 code smell 的重构技术。

关键词：code smell，软件维护，重构

1 背景介绍

软件维护是开发过程中的一项重要工作。大概 50%~80%的软件成本均与维护活动有关：修复设计和实现故障、是软件适应不同环境以及添加或修改功能。大型和复杂软件项目中的 code smell 通常给软件维护带来很多负面的印象。code smell 违反了编码设计原则，增加了潜在的技术债，并可能引入缺陷。通常开发人员在修改或添加功能以满足新的软件需求时很容易引入 code smell，这使得代码变得复杂难懂，并且降低了软件质量。图 1 列举了 22 种 code smell 并给出了可能的重构方案。

S No.	Name of smell	Solution
1.	Duplicated Code	*Extract Method, Pull Up Method, Substitute Algorithm.*
2.	Long Method	*Extract Method, Replace Temp With Query, Introduce Parameter Object and Preserve Whole Object. Replace Method with Method Objects, Decompositional Objects*
3.	Large Class	*Extract Class, Extract Sub Class. Duplicate Observed Data*
4.	Long Parameter List	*Replace Parameter with Method*
5.	Divergent Change	*Extract Class*
6.	ShotGun Surgery	*Move Class or Move Method*
7.	Feature Envy	*Move Method and Extract Method*
8.	Data Clumps	*Extract Class, Introduce Parameter Object and Preserve Whole Object*
9.	Primitive Obsession	*Replace Data Value With Object, Replace Type Code With Class, Replace Type Code With Subclass., Replace Type Code With State Strategy, Extract Class, Introduce Parameter Object And Replace Array With Object.*
10.	Switch Statements	*Extract Method, Move Method, Replace Type Code with Subclass or Replace Type Code with State*
11.	Parallel Inheritance Hierarchies	*Move Method or Move Field*
12.	Lazy Class	*Collapse Hierarchy or Inline Class*
13.	Speculative Generality	*Collapse Hierarchy, Inline Class, Remove Parameters Methods, Remove Methods*
14.	Temporary Fields	*Extract Class, Introduce Null Objects*
15.	Message Chains	*Hide DELEGATE, Extract Method and Move Method.*
16.	Middle Man	*Remove Middle Man, Inline Method, Replace Delegation with Inheritance*
17.	Inappropriate Intimacy	Move Method or Move Field, Change Bidirectional Association to Unidirectional, Extract Class, Hide Class
18.	Alternative Classes with different interfaces	*Rename Method, Move Method, Extract Superclass*
19.	Incomplete Library Class	*Move Method, Introduce Foreign Method, Introduce Local Extension*
20.	Data Class	*Encapsulate Field or Encapsulate Collection, Remove Setting Method, Move Method or Extract Method, Hide Method*
21.	Refused Bequest	*Push Down Method Push Down Field, Replace Inheritance with Delegation*
22.	Comments	*Extract Method or Rename Method, Introduce Assertion.*

图 1

这些 code smell 可以分为以下 7 大类，见图 2

S No.	Unit	Code smells involved
1	Bloaters	Long Method, Large Class, Data Clumps, Primitive Obsession and Long Parameter List
2	Object Orientation Abusers	Switch Statements, Temporary Field, Refused Bequest and Alternative Classes with Different Interfaces
3	Change Preventers	Divergent Change, Shotgun Surgery and Parallel Inheritance Hierarchies
4	Dispensables	Lazy class, Data class, Duplicate Code, Dead Code and Speculative Generality
5	Encapsulators	Message Chains and Middle Man
6	Couplers	Feature Envy and Inappropriate Intimacy
7	Others	Incomplete Library Class and Comments

图 2

1. The Bloaters：单个代码实体规模过大、难以管理，比如过长函数、过大的类、过长参数列表等
2. The Object-Orientation Abusers：违反面向对象编程原则的设计，比如临时字段
3. The Change Preventers：软件设计不当导致后续需求需要修改代码时很容易引入错误或者难以正确地修改

4. The Dispensables：代码中包含冗余结构或元素
5. The Encapsulators：数据封装的不当使用
6. The Couplers：违反模块内高内聚、模块间低耦合的代码
7. Others：除了以上 6 类的其他 code smell，比如不完整清晰的代码注释

重构可以去除 code smell，重构是一种不改变系统外部行为并且改善系统内部结构的行为。

2 重构技术

根据检测 code smell 的技术方法不同，将相关技术分为 3 类：基于可视化的方法、半自动检测方法、自动检测方法。以下将分类别介绍一种去除 code smell 的重构技术。

2.1 基于可视化的方法

Fowler 在其关于重构的论述著作中谈论了针对不同类型的 code smell 的重构方法，但是这些重构很多是基于人类的直觉，Simon 提出应该基于代码静态结构分析和度量的可视化来辅助开发人员识别出系统中应该重构的部分，Simon 的重构方法主要有以下 4 类：

1. 移动方法：将某一方法移动到更常被调用的类
2. 移动属性：将某一属性移动到更常使用的类
3. 提取类：创建一个新类，并且将一些内聚的属性和方法从旧类移动到新类
4. 内联类：将一个类中的所有成员移动另一个类中，然后删除旧类

以上这些重构方法都是基于“高内聚，低耦合”这一设计原则的，为了应用这些重构方法，针对不同代码实体的相似性度量是十分必要的。Simon 认为两个实体的相似性与它们共享的实体集有关。假设𝔹是计算相似性所需要考量的属性集合，可以定义两个实体间的距离度量如图 3 下：

$$dist_{\mathbb{B}}(x,y) := 1 - \frac{|p(x) \cap p(y)|}{|p(x) \cup p(y)|}$$

图 3

其中 p(x)表示实体 x 具有的所有属性。这里属性集合𝔹的定义是十分重要的，因为相似性的计算很大程度上依赖于属性集合的设定，不同的属性集合可能导致相似度计算相差较大。由此 Simon 提出了一个通用度量框架，主要有以下两步：

1. 确定应测量的实体类型
2. 确定计算相似性应考虑哪些属性

下面介绍一个进行移动方法重构的示例：移动方法的重构场景识别。

移动方法的重构考虑的实体是方法，由于方法内有方法调用和数据成员访问，由此我们可以定义集合𝔹为方法实体所使用的方法和数据成员集合，针对每一个实体，我们定义该实体在相似性度量下的属性集合如表 1 所示：

方法	数据成员
方法自身	数据成员本身
方法内直接调用的方法	所有直接使用该数据成员的方法
方法内直接使用的数据成员	

表 1

上述集合𝔹和相似性度量的设计是基于以下原则：一个类的方法应该更多地使用本类的方法和数据成员。下面给出一个基于相似性度量可视化的示例。

```
class class_A
{
public:
     static void methodA1()
     {    attributA1=0;
          methodA2();}
     static void methodA2()
     {    attributA2=0;
          attributA1=0;}
     static void methodA3()
     {    attributA1=0;
          attributA2=0;
          methodA1();
          methodA2();}
     static int attributA1;
     static int attributA2;
}
```

```
class class_B
{
public:
     static void methodB1()
     {    class_A::attributA1=0;
          class_A::attributA2=0;
          class_A::methodA1();}
     static void methodB2()
     {    attributB1=0;
          attributB2=0;}
     static void methodB3()
     {    attributB1=0;
          methodB1();
          methodB2();}
     static int attributB1;
     static int attributB2;
}
```

图 4

我们可以看到图 4 中的 class_B 中的 methodB1 具有 code smell，因为其只使用 class_A 的方法和数据成员。为了识别出这种 code smell，图 5 给出了计算每两个实体（六个方法和四个数据成员）的距离度量，图 6 是使用可视化技术生成的实体距离模型，其中绿色图形为 class_A 的元素，蓝色图形为 class_B 的元素，所有数据成员均为正方体，所有方法均为球体。

	mA1()	mA2()	mA3()	mB1()	mB2()	mB3()	aA1	aA2	aB1	aB2
mA1()	0									
mA2()	0.5	0								
mA3()	0.4	0.	0							
mB1()	0.6	0.6	0.5	0						
mB2()	1	1	1	1	0					
mB3()	1	1	1	0.86	0.6	0				
aA1	0.5	0.67	0.33	0.5	1	0.88	0			
aA2	0.83	0.6	0.5	0.67	1	0.86	0.5	0		
aB1	1	1	1	1	0.5	0.25	1	1	0	
aB2	1	1	1	1	0.33	0.8	1	1	0.75	0

图 5

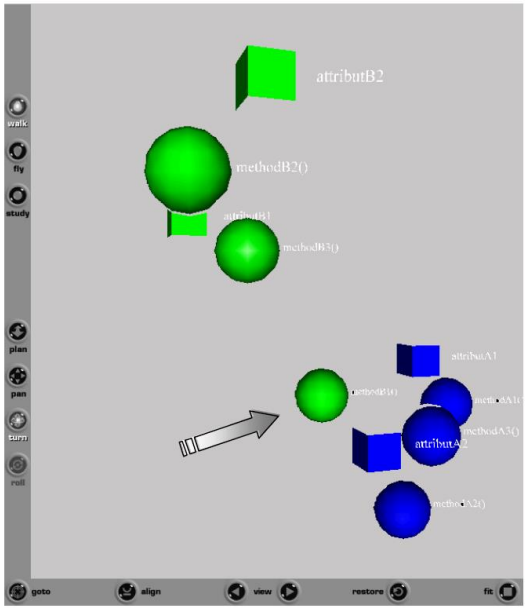

图 6

可以看到 methodB1 实体更靠近 class_A 中的实体，所以可以将 methodB1 移动到 class_A 中进行移动方法的重构。

2.2 半自动检测方法

Liu 提出了一种新的针对 Java 代码的半自动泛化重构方法。泛化重构通过继承关联类和共享函数。要启用泛化重构，开发人员应首先确定潜在的泛化重构机会。对于复杂的软件系统而言，人工识别出这些机会是非常困难且耗时的。为此 Liu 设计了一个工具 GenReferee，其工作流程图如图 7 所示。通过类的概念关系、类实现的相似性以及继承关系来识别出可能的泛化重构机会。

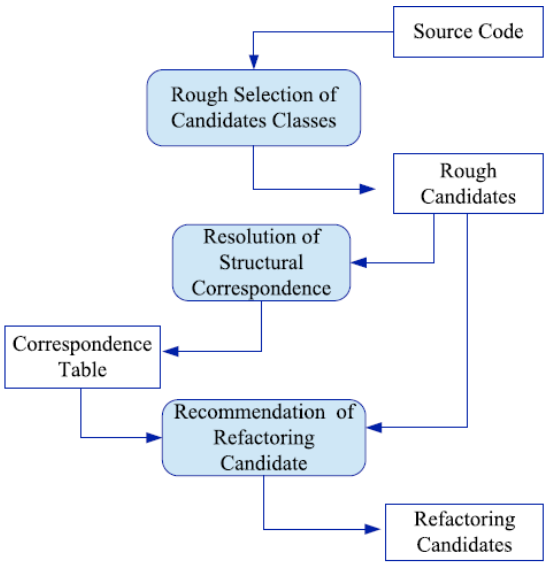

图 7

GenReferee 主要以以下 3 个阶段进行:

1. GenReferee 使用快速选择算法对整个项目源代码进行候选类的粗略选择，成对的候选类作为 Rough Candidates
2. 对于 Rough Candidatas 中的每对候选类，GenReferee 解析其结构对应关系。
3. 根据每对候选类的结构对应关系，GenReferee 推荐一系列的重构机会

算法的核心部分在于解析候选类的结构对应关系，第一阶段的粗略选择主要是为了提高整体算法的性能，这是因为在复杂的软件项目中，类的成对组合数量非常大。而且由于解析每对候选类之间的结构对应关系非常耗时，所以算法的第一阶段进行粗略选择，然后在随后的阶段中更严格地检查所有候选对，这样不仅能提高 GenReferee 的效率，还提高的方法的精度。

粗略选择阶段根据类的声明和实现来选择候选类。继承关系能够将同一概念类别的类关联起来以共享公共接口，但是如果两个相似的类共享一些重复或相似的源代码，但是它们之间没有概念上的联系，那么委托关系是删除这些重复代码的正确方式，在这种情况下使用继承会造成类层次结构混乱且难以理解。

GenReferee 首先从类名、属性名、方法名和参数名中提取名词和动词来选择概念上相关的类。GenReferee 假设定义的标识符名称遵循的是驼峰式写法，然后根据大写字母提取其中的单词进行分析。GenReferee 也会根据类实现上的相似性来进行选择类选择，这时可以通过代码克隆检测工具来进行相似性比较。

对于第一阶段选择的每对候选类，GenReferee 会比较候选类的数据成员和方法，数据成员只有在具有相同类型和名称的情况下才能匹配，其中类型的比较不考虑继承的层次关系。方法的比较要复杂些，两个方法间的相似度用以下公式计算：

$$S_m(m_i, m_j) = \frac{1}{2} \times [S_{sg}(m_i, m_j) + S_{imp}(m_i, m_j)]$$

其中$S_m(m_i, m_j)$是方法m_i和方法m_j的总相似度，S_{sg}和S_{imp}分别是函数签名相似度和函数实现相似度。

计算出数据成员和方法的相似度之后，GenReferee 会根据相似度表对每对候选类计算推荐强度值，并推荐一系列重构方式。根据候选类对(C_i, C_j)在类继承层次结构的位置，可以分为以下四类：

1. C_i和C_j都没有除 Object 类之外的其他父类，此时可以通过引入一个父类来共享它们的数据成员和方法
2. C_i和C_j具有相同的直接父类，此时可以将它们相同的数据成员和方法提取到其共同父类中
3. C_i和C_j是不相关的类，即C_i和C_j除了 Object 外没有共同祖先，由于 Java 不支持多重继承，此时可以定义一个接口共享功能
4. C_i和C_j间接继承于同一祖先（不是 Object）,此时也将它们的通用方法提取到同一接口

对于每对候选类的推荐强度和推荐的重构方式，开发者可以手动逐个检查这些信息然后决定是否进行重构。

2.3 自动检测方法

现代 IDE 为很多简单的重构提供了一些自动化支持，但是没有提供“提取类”重构方式。Marios 提出了一种方法用于识别一些内聚性更强的属性集合来分解软件系统中规模较大的模块，从而改善整体系统设计。整个工作流程分为以下三步：

1. 寻找提取类的重构机会
2. 对这些提取类的重构方式评估对系统带来的预期改进
3. 自动化进行提取类重构

图 8 给出了一个 Java 的 Person 类定义，下面将以该类作为示例介绍工作流程。

```
public class Person {

    private String name;
    private String job;
    private String officeAreaCode = "555";
    private String officeNumber;

    public Person(String officeNumber) {
        this.officeNumber = officeNumber;
    }


    public String getTelephoneNumber() {
        String phone = officeAreaCode + "-" + officeNumber;
        name += ", " + phone;
        job += ", " + phone;
        return phone;
    }

    public void changeJob(String newJob) {
        if(!newJob.equals(job)) {
            this.job = newJob;
        }
        name += ", "+newJob;
    }

    public void modifyName(String newName) {
        if(!newName.equals(name)) {
            this.name = newName;
        }
        job = newName + ", " + job;
    }
}
```

图 8

第一阶段寻找提取类的重构机会分为两步，首先计算每个类中的成员之间的距离，然后使用聚类算法识别出能够作为提取到新类的内聚性较高的聚类，然后评估这些构造的新类是否能够保留系统原来的行为且改善设计。距离度量算法是计算类成员间的 Jaccard 距离，为了定义两个成员之间的 Jaccard 距离，采用了实体集概念。其中数据成员的实体集包含：数据成员本身，被直接访问的方法、通过 public 的 get 或 set 方法访问该成员的方法；方法成员的实体集包含：方法本身、方法访问的数据和方法。定义了成员的实体集之后，定义成员α和β的 Jaccard 距离为：

$$d_{\alpha,\beta} = 1 - \frac{|A \cap B|}{|A \cup B|}$$

其中 A 和 B 分别表示α和β的实体集，图 9 显示了 Person 类成员间的 Jaccard 距离

	name	job	officeAreaCode	officeNumber	changeJob()	modifyName()
job	0.4					
officeAreaCode	0.8	0.8				
officeNumber	0.8	0.8	0.67			
changeJob()	0.6	0.6	1	1		
modifyName()	0.6	0.6	1	1	0.5	
getTelephoneNumber()	0.71	0.71	0.6	0.6	0.67	0.67

图 9

利用 Jaccard 距离和聚类算法可以得到聚类图，如图 10 所示：

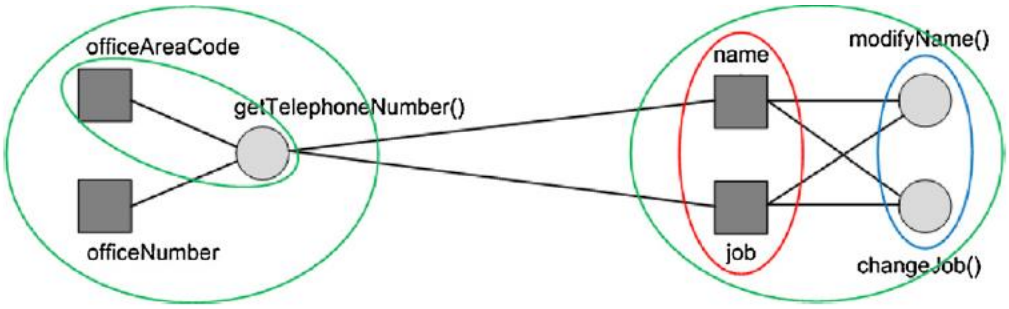

图 10

其中正方形节点代表数据成员，圆形节点代表方法，结点间的边表示成员间存在依赖关系，边的长度与成员间的 Jaccard 距离成正比，利用分层聚类算法可以得到图 11。

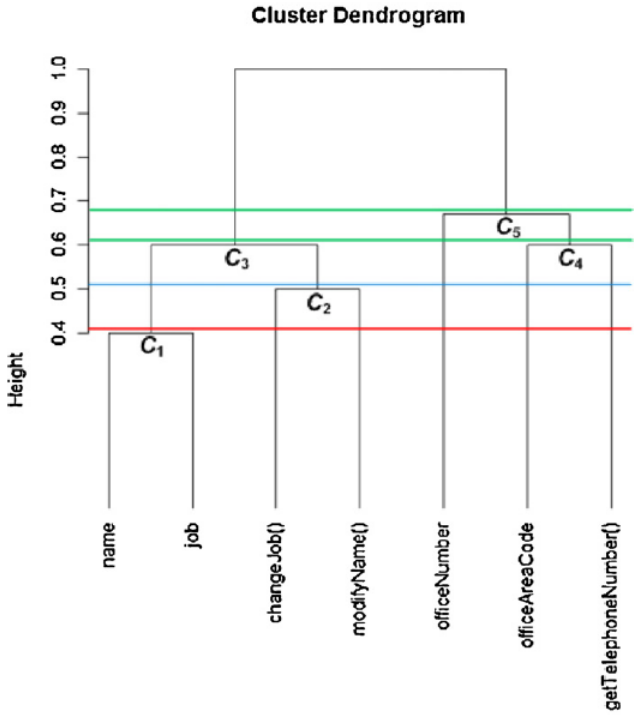

图 11

可以看到通过聚类算法得到 5 个簇：

1. C1 = {name, job}

2. C2 = {modifyName(), changeJob()}
3. C3 = {name, job, modifyName(), changeJob()}
4. C4 = {officeAreaCode, getTelephoneNumber()}
5. C5 = {officeAreaCode, officeNumber, getTelephoneNumber()}

然后看到 C1、C2、C4 都是可能的实施提取类的重构机会，然而通过评估去掉不合理的重构机会，聚类生成的新类中如果出现以下 5 种情况即不能被提取：

1. 包含抽象方法
2. 包含可见性大于 private 且被其它类使用的成员
3. 包含重载了抽象方法的方法
4. 仅包含数据成员
5. 包含调用了父类方法的方法

第二阶段将对剩下的 C2 和 C4 新类进行评估，使用 Entity Placement 度量，该度量计算一个类中的实体与类本身的距离（内聚性）除以不属于该类的实体与类本身的距离（耦合性），综合考虑了系统的耦合性和内聚性。在执行提取类重构过程中将会改善系统的内聚性，恶化耦合性。如果重构过程中内聚性的改善明显大于耦合的恶化，即 Entity Placement 值明显降低，则可以执行重构。图 12 显示了分别提取 C2 类和提取 C4 类引起的系统 Entity Placement 变化

Refactoring Type	Source Class/General Concept	Extractable Concept	Entity Placement
1	Person		0.572655407227616
2	offic + number		
Extract Class	3	offic + number	0.572655407227616
2	name + job		
Extract Class	3	name + job	0.6310535257021981
	current system		0.6995797637590863

图 12

可以看到提取 C4 后系统 Entity Placement 值明显降低，所以我们选择将 C4 中的成员提取成为一个新类。

第三阶段使用 Eclipse 的 JDT 进行重构，首先从源码构建出 AST，然后基于提取类的重构机制修改 AST，然后重写代码。最终将 C4 类提取成新类 TelephoneNumber，类定义如图 13 所示：

```java
public class TelephoneNumber {

    private String officeAreaCode = "555";
    private String officeNumber;

    public String getOfficeAreaCode() {
        return officeAreaCode;
    }

    public void setOfficeAreaCode(String officeAreaCode) {
        this.officeAreaCode = officeAreaCode;
    }

    public String getOfficeNumber() {
        return officeNumber;
    }

    public void setOfficeNumber(String officeNumber) {
        this.officeNumber = officeNumber;
    }

    public String getTelephoneNumber(Person person) {
        String phone = officeAreaCode + "-" + officeNumber;
        person.setName(person.getName() + ", " + phone);
        person.setJob(person.getJob() + ", " + phone);
        return phone;
    }
}
```

图 13

Person 类进行重构前后的对比如图 14 所示：

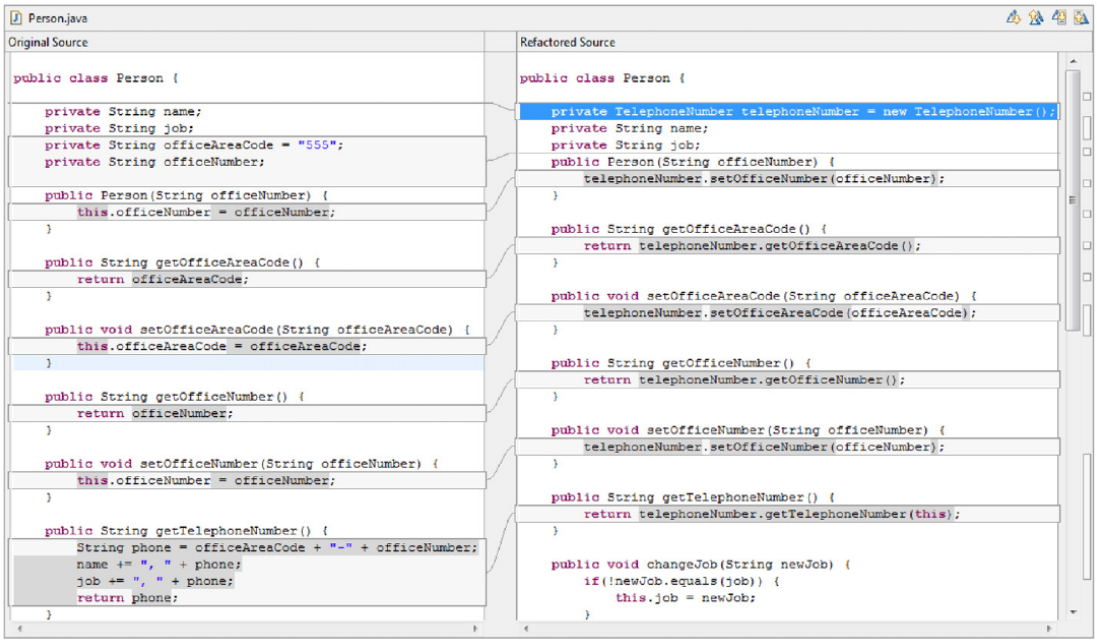

图 14

3 总结

本文介绍了 code smell 和重构的基本概念，并且介绍了若干面向对象系统中去除代码 code smell 的检测和重构技术，可以发现对类成员间进行合适的距离度量或者相似度比较是因系统而异的，而针对不同 code smell 的重构方案现已比较成熟且趋同。因此针对不同 code smell 的检测提出合适的度量手段是十分重要的。

参考文献

[1]Simon F, Steinbruckner F, Lewerentz C. Metrics based refactoring. In: Proc. fifth european conf. software maintenance and re-eng; 2001. p. 30.
[2]Liu Hui, Niu Zhendong, Man Zhiyi, Shao Weizhong. Identification of generalization refactoring opportunities. Autom Softw Eng 2013;20 (1):81–110.
[3]M. Fowler.Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999
[4]T. Mens and T. Tourwé. A survey of software refactoring.IEEE Transactions on software engineering, 30(2):126–139, 2004.
[5]E. Murphy-Hill and A. Black. Making Refactoring Tools Part of the Programming Workflow. 2008a.
[6]E. Murphy-Hill and A. Black. Refactoring tools: Fitness for purpose.IEEE software, 25:38–44, 2008b.
[7]E. Murphy-Hill, C. Parnin, and A. Black. How we refactor, and how we know it. InInternationalConference on Software Engineering 2009, volume 2, pages 0–24, 2009
[8] Fokaefs Marios, Tsantalis Nikolaos, Stroulia Eleni, Chatzigeorgiou Alexander. Identification and application of Extract Class refactorings in object-oriented systems. J Syst Softw 2012;85(10):2241–60.