基于设计变动分析的JAVA源代码变动预测

谭向臣,冯 铁,罗术通,李大利

(吉林大学 计算机科学与技术学院,长春 130012)

摘 要:提出了一种映射面向对象软件设计变动到程序代码变动的方法来预测可能产生的源代码变动。该方法把设计与代码都定义和分类为原子变动与复合变动,根据变动的定义与分类提出并应用变动影响分析的算法,从而得出从UML设计模型变动到Java代码变动的映射规则。并且设计了一个支撑工具,对一个开源软件进行分析来验证本文的方法。

关键词:计算机软件;变动影响分析;面向对象软件设计;Java编程语言

中图分类号:TP311 **文献标识码**:A **文章编号**:1671-5497(2008)03-0685-05

Java source code change prediction based on design change analysis

Tan Xiang-chen, Feng Tie, Luo Shu-tong, Li Da-li

(*College of Computer Science and Technology, Jilin University, Changchun* 130012, *China*)

Abstract: An approach of mapping object-oriented design changes to program changes are proposed, so that the prediction of possible source code change effort can be made. This approach focus on the definition and taxonomy of atomic changes and composite changes at both level, according to which change impact analysis algorithms are proposed and applied to conclude the change mapping rules from UML based design model to Java source code. A supporting tool implementing the heuristics is designed and a case study on an open source project is conducted to validate our method.

Key words: computer software; change impact analysis; object-oriented software design; Java programming language

为了解决由于软件变动需要根据候选设计的复杂性和变动成本[1],以及所能引起的潜在的影响(side effect)来评价一个设计的优劣的问题,作者基于对设计变动及其变动影响分析[2]提出了一种对设计变动及其可能产生的代码变动进行预测的方法。

1 设计变动与分类

UML应用范围很广,但是并不是其所有的语言特性在所有的应用领域中都是必要的,要对此语言进行适当模块化,选择有直接关系的部分[3]。文献[4]中提供了概要设计与详细设计的说明,本文中主要考虑一些属于概要设计的UML设计元素的变动。

1.1 原子设计变动

只考虑UML设计元素变动本身而不考虑其可能产生的影响,这样的变动视为原子设计变动。

类之间的关系也作为原子设计变动来考虑,

收稿日期:2007-03-05.

基金项目:"863"国家高技术研究发展计划专题课题(2007AA01Z123).

作者简介:谭向臣(1982-),男,硕士研究生. 研究方向:软件设计改进. E-mail:xiangchen-tan@sohu.com

通讯联系人:冯铁(1972-),男,副教授. 研究方向:基于事例的推理,软件体系结构,软件复用技术,面向对象技术. E-mail:fengtie@jlu.edu.cn

考虑四种 UML 可以表示的类之间的关系,分别为:关联(A)、泛化(G)、依赖(D)、实现(R)。

用 r (B, C)表示类 B 和类 C 之间存在关系 r,那么 D (B, C)就表示类 B 依赖类 C。G (B, C) 表示类 B 继承类 C,R (B, D) 表示类 B 实现了接口 D,A (B, C)表示类 B 与类 C 存在关联关系。

用下面这个四元组表示原子设计变动:

(CT,Tag,CE,ATC)

(1)CT 表示变动类型,分别为增加(add)删除(delete)变动(change) 。

(2)Tag 是一个标记,表示变动元素的类型,分别为:方法("M"), 属性("A"),类("C"),关系("R")。

(3)CE 表示具体的变动元素,根据所表示变动元素的类型不同而不同。如果 Tag 的值是"M",那么 CE 是一个二元组(类名,方法名),Tag 的值是"C", CE 则是类的名字。Tag 的值是"A",CE 是(类名,属性名)。Tag 的值是"R",CE 是四种 r (B, C)之一。

(4)ATC 表示一个 CE 的所有属性,当 CT=change 时, 每个属性由一个二元组来表示(属性变动前,属性变动后),如果相应 CE 没有属性,则 ATC 可以为空,用 null 表示空值,当 CT = add 或 delete 时,每个属性只是一个单值,表示 CE 的当前或结果属性。ATC 中有哪些属性根据以下的说明加以区别:

(1)方法:可见性、抽象性、静态、返回类型,参数表。

(2)属性:可见性、静态性、可变性、类型。

(3) 类:可见性、抽象性、可变性。

(4)关系:泛化、实现和依赖关系没有属性,其 ATC 为空。关联的属性体现在两个关联端上,每个关联端按顺序有可见性、多元性、可变性、可导航性 4 个属性。

(add/delete, Tag, CE, ATC)表示增加/删除一个类型为 Tag 的元素 CE, CE 具有 ATC 中的属性。

(change, Tag, CE, ATC)表示改变一个类型为 Tag 的元素 CE 的一个或多个属性。

1.2 复合设计变动

Haney[5]用波动效应(ripple effect)来描述一个模块中的一个变动会引发其他模块一些必要的变动过程。利用波动效应定义设计上的复合变动。

定义 1 C_0 表示一个给定的原子设计变动,$C_i \Rightarrow C_j$ 表示原子设计变动 C_i 可以引发原子设计变动 C_j,关于原子设计变动 C_0 的复合设计变动 A 是一个原子设计变动集合,并且 A 满足:

(1) $C_0 \in A$;

(2) $\forall C_i \in A$, $C_i \Rightarrow C_j$, $C_j \in A$。

下面给出计算关于某个原子设计变动的复合设计变动的算法:

(1) Composite (C)

输入:原子设计变动 C

输出:关于 C 的复合设计变动 A。

```
Begin
1. A: = {(C, True)};
    //原子变动和是否做变动分析的标记
2. FOR each Ci∈A and Tag == True DO
Begin
Tag: = false;
FOR each Cj∈ Changes(Ci)
A = A∪Composite (Cj);
End
Return A;
End
```

(2) Changes (C_i)

输入:原子设计变动 C_i。

输出: 根据不同的原子设计变动类型,选择不同的算法以返回由原子设计变动 C_i 直接引发的原子设计变动的集合。

```
Begin
CS :={};
Switch (Ci)
/*每次只对一种属性的变动进行影响分析*/
Case (add, "M", (class name , method name), [null, abstract, null, null, null]):
CS = addAbstractMethod (class name, method name); Break;
Case (….): CS = method (…);
Break; ……
Return CS;
End
```

以下是计算复合设计变动算法的一个例子:

(3) addAbstractMethod (P, M)

输入:类名、方法名。

输出:由于增加抽象方法可直接引发的原子设计变动集合。

```
Begin
```

1. CS：= {}；

2. If (Class "P" is not abstract)

CS：= CS∪(change, "C", "P", [(no change), (～abstract, abstract), (no change)])；

3. for (each class PS of P's son)

If (PS is not a leaf class)

CS：= CS∪addAbstractMethod (PS, M)；

Else

CS：= CS∪(add, "M", (PS, M), […])；

Return CS；

End

针对每一种原子设计变动类型，均给出计算其可能直接引发的原子设计变动集合的算法。

2　代码变动与分类

2.1　原子代码变动

参考 X. Ren[6] 中提出的代码的原子变动分类，并对其进行了一些扩展。表 1 列出了原子代码变动分类。其中前 4 种是参考文献[6]中提出的，另外扩展了后 7 种分类，用表格中的格式来

表 1　原子代码变动分类表

Table 1　Categories of atomic code changes

AF (class name, Name, Type, [visibility | changeability])	Add field (增加域)
DF (class name, Name, Type, [visibility | changeability])	Delete field (删除域)
AC (name, [visibility | abstraction | changeability])	Add empty class (增加空类)
DC (name, [visibility | abstraction | changeability])	Delete empty class (删除空类)
CC (name, [visibility | abstraction | changeability | generalization])	Change Class (变动类)
AMD (class name, method name, [abstraction | visibility | static | signature])	Add method declaration (增加方法声明)
AMB (class name, method name)	Add method body (增加方法体)
DMD (class name, method name, [abstraction | visibility | static | signature])	Delete method declaration (删除方法声明)
DMB (class name, method name)	Delete method body (删除方法体)
CMD (class name, method name, [abstraction | visibility | static | signature])	Change method declaration (变动方法声明)
CMB (class name, method name)	Change method body (变动方法体)
CF (name , Type, [Visibility | changeability])	Change field (变动域)

表示相应变动元素，[ ]中是相应元素的属性，在实例中只表示被变动的属性变动后的结果。如：CC (B, [abstract]) 表示只将 B 的抽象性变动为抽象的。

2.2　复合代码变动

从设计变动得出的代码变动没有考虑代码变动之间固有的但不能体现在设计中的依赖关系。所以根据代码之间的依赖关系，定义复合代码变动如下：

定义 2　C_0 表示一个给定的原子代码变动，$\prec^*$ 表示原子代码变动之间的偏序关系，$C_j \prec^* C_i$ 表示原子代码变动 C_j 是原子代码变动 C_i 的先决条件，即 C_j 必须发生在 C_i 之前。关于原子代码变动 C_0 的复合代码变动 B 是一个原子代码变动集合，并且 B 满足：

(1) $C_0 \in B$

(2) $\forall C_i \in B, C_j \prec^* C_i, C_j \in B$

3　映射原子设计变动到代码变动

本节中，body (M) 表示方法 M 有方法体。methods (A) 表示类 A 或者接口 A 的所有方法的集合。attri (A) 表示类 A 的所有属性的集合，type (a) 表示 a 的类型，Sig (M) 表示方法 M 的 signature 的类型集合。

3.1　类自身变动到代码变动

这里考虑类自身的属性：可见性，抽象性，可变性。如果在设计中类 B 的可变性变为只读的 (read-only)，在代码中就一定要在类 B 的声明中加入关键字"final"如(1)所示：

(change, "C", B, [, , (～read-only, read-only)])

$$\Rightarrow CC(B, [\text{final}]) \quad (1)$$

"⇒"表示设计与代码间的映射关系，设计变动中的 ATC 并未写全，只写了与变动相关的部分，未变动的部分省略未写，下同。

在(2)中类 B 变为了抽象类，则在类 B 的代码声明中要加入"abstract"关键字。

(change, "C", B, [, (～abstract, abstract),])

$$\Rightarrow CC(B, [\text{abstract}]) \quad (2)$$

可见性的变动由于篇幅所限未给出。类的名字的变动则看作是删除一个旧类再增加一个新类。

3.2　类内结构元素的变动到代码变动

可变动的类内结构元素有类的属性与方法。此处只考虑基本数据类型的属性，它们在设计与

代码中是一一对应的。至于属性的类型是类的情况则视为关联。在设计中可以增加、删除方法和变动方法的属性。增加和删除方法可以映射到代码中类的声明与实现部分,如(3)和(4)所示(其中方法的属性在设计与代码中一一对应,未标出):

$$(\text{add},\text{“M”},(B,m),[\ldots]) \Rightarrow \text{AMD}((B,m),[\ldots]) \wedge \text{AMB}(B,m) \quad (3)$$

$$(\text{delete},\text{“M”},(B,m),[\ldots]) \Rightarrow \text{DMD}((B,m),[\ldots]) \wedge \text{DMB}(B,m)(\text{body}(m)) \quad (4)$$

如果在设计中一个方法从一个非静态方法变动为一个静态方法,那么这个方法的方法体在实现时将不能再存取类中的非静态的属性。具体规则由于篇幅所限略。可见性的变动与类自身可变性的变动相似,不再举例。方法名的变动与类名变动方法相同。方法签名在设计与代码中也是一一对应的。

3.3 类之间关系变动到代码变动

(1)增加/删除泛化

增加泛化关系 G (B, C),如(5)所示,如果类 C 是一个非抽象类,那么只需代码中增加简单继承。如果类 C 是抽象类,要在增加泛化关系的同时在类 B 中增加类 C 中的抽象方法,实现与否取决于类 B 是否是抽象类。

$$(\text{add},\text{“R”},G(B,C),[\text{null}]) \Rightarrow \begin{cases} \text{CC}(B,[\text{extends}(C)]) \\ \quad (\forall m \in \text{methods}(C) \wedge \sim \text{abstract}(m)) \\ \text{CC}(B,[\text{abstract, extends}(C)]) \\ \quad \wedge \text{AMD}(B,m,[\ldots]) \wedge \text{AMB}(B,m) \\ (\exists m \in \text{methods}(C) \wedge \text{abstract}(m) \wedge m \notin \text{methods}(B)) \\ \text{CC}(B,[\text{extends}(C)]) \wedge \text{AMD}(B,m,[\ldots]) \\ \wedge \text{AMB}(B,m) \\ \quad (\forall m \in \text{methods}(C) \wedge \text{abstract}(m)) \end{cases} \quad (5)$$

删除泛化关系在代码中会产生很大影响,但由于对泛化后的使用都在方法体的实现中,并不能在类图中体现出来。

(2)增加/删除实现

$$(\text{add},\text{“R”},G(B,C),[\text{null}]) \Rightarrow \text{AMD}((B,m),[\ldots]) \wedge \text{AMB}(B,m)^* (\forall m \in \text{methods}(D)) \quad (6)$$

实现关系是类和接口之间的关系,如(6)所示,增加一个实现关系 R (B, D),类 B 就要声明和实现接口 D 的所有方法。而删除实现关系 R (B, D)同删除一个泛化关系相似。

(3)增加/删除关联

类图中增加一个双向关联 A (B, C) 将导致在 A 类中增加一个类 B 的类型的域并且在 B 类中增加一个类 A 的类型的域,其相应属性由类图中的关联端的属性决定,如(7)。

$$(\text{add},\text{“R”},A(B,C)),[B(,,,\text{true}),C(,,,\text{true})] \Rightarrow \text{AF}(B,\text{“c”},C,[\ldots]) \wedge \text{AF}(C,\text{“b”},B,[\ldots]) \quad (7)$$

删除一个双向关联 A (B, C),其效果与增加一个双向关联相反,需要删除相应类型的域。

(4)变动关联

变动两个类之间的关联关系取决于对关联端属性的变动,如 (8)所示,

$$(\text{change},\text{“R”},A(B,C)),[B(,,,(\text{true},\text{false})),C(,,,(\text{true},\text{ture}))] \Rightarrow \text{DF}(C,\text{“b”},B,[])^* (\forall b \in \text{attri}(C) \wedge \text{type}(b) = B) \quad (8)$$

关联 A (B, C)从双方向(B↔C)变为单方向(B－＞C),类 C 中属性为 B 类型的属性将被删除。(8)只是变动关联端属性的一个例子。变动靠近关联端 B 的可见性与可变性将会导致相应的类 C 中的属性类型为 B 类型的属性的相应变动。对依赖的增加删除由于篇幅所限未列出。

4 映射复合设计变动到代码变动

本节给出复合设计变动与代码变动之间的关系。A 是设计变动(CT,Tag,CE,ATC)的复合设计变动。为了完成整个变动关系的映射,分为 4 个步骤:

(1)分解(CT,Tag,CE,ATC)为原子设计变动集合 S. $\forall C_i \in S, C_i$ 在 ATC 中只含有一个变动的属性元素。

(2)$\forall C_i \in S, A_i = \text{Composite}(C_i)$ A 是复合设计变动。$A = A_1 \cup A_2 \cup \cdots A_i \cup \cdots \cup A_n (1 \leqslant i \leqslant n, n = |S|)$

(3)$\forall C_i \in A, C_i \Rightarrow ACC_i^*$, $Z = Z \cup \{ACC_i^*\}$, ACC_i^* 原子代码变动,“*”表示 ACC 可能不只一个元素。Z 是原子代码变动集合。

(4)$\forall ACC_i \in Z, \exists \beta \notin Z \wedge \beta <^* ACC_i, Z = Z \cup \{\beta\}$,$\beta$ 表示一个原子代码变动,那么 Z 就是映射复合设计变动到代码变动的结果集。

5 代码变动预测工具(CCPTool)

图 1 展示了代码变动预测工具(CCPTool)的

架构,由三部分组成:原子设计变动检测模块(ADCD),复合设计变动抽取模块(CDCE)和规则映射模块(MR)。

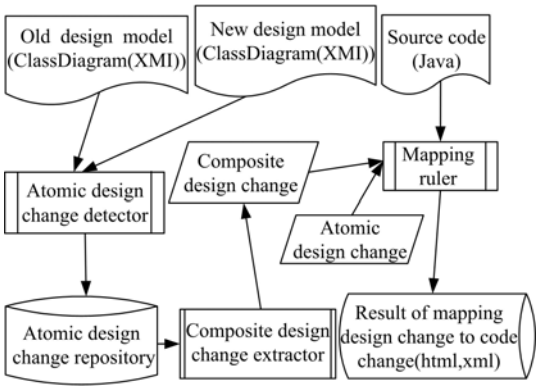

图 1　CCPTool 的架构

Fig. 1　Architecture of CCPTool

ADCD 模块解析描述系统新旧设计的 XMI(XML 元数据交换)文件,比较两个设计的 XMI 文件,获取其中所有的原子设计变动。Atomic design change repository 用来保存由 ADCD 产生的原子设计变动。CDCA 模块从库中读取原子设计变动并根据复合设计变动算法和影响分析规则抽取出相应的复合设计变动和原子设计变动。最后,由 MR 模块依据设计到代码的映射规则给出新设计相对于旧设计在代码上可能会产生的变动。

6　案例分析

选择开源软件 XUI RIA Framework[7] 作为分析案例,XUI 是一个构建灵活 Java 和 XML 程序的 RIA 平台。比较 XUI 两个相邻的版本如表 2 所示。考虑两个版本的软件包 net. xoetrope. swing 的变动。

表 2　两版本比较结果

Table 2　Result of compare two versions

	Class	Files	Code lines	size
V2.0.5	313	276	29716	1.67 M
V2.0.6	354	306	33523	1.95 M

从表 3 中可以看到,除了方法体的变动外,本文的方法可以有效预测实际代码变动。从以上分析发现,继承关系的变动和单纯算法的改进会引发方法体的变动,但由于设计上没有关于方法体的实现,所以在设计上得不到这些变动。有些方法体的变动由于复合代码变动的原因也是可以预测到的,但只是预测其会变,而预测不到其具体会怎样变。

表 3　实际代码变动与预测得到的代码变动

Table 3　Source code change VS predict code change, CR(change relationship)

	AC	CMD	CMBCF or AF	AM	DM	CR
Practice	3	2	24	27	31	3	3
Predict	3	2	3	27	31	3	3

7　结束语

提出了一种基于 UML 设计分析预测源代码变动的方法。将设计与代码变动都分类为原子与复合变动,并且将设计元素的变动与代码的变动做了映射,用变动影响分析的方法来预测代码的变动。并用一个开源程序进行了验证,本文提出的方法具有一定的预测作用。

参考文献:

[1] Jussi Koskinen. Software maintenance costs[EB/OL]. http://www.cs.jyu.fi/~koskinen/smcosts.htm,2004.

[2] Bohner S A, Arnold R S. An introduction to software change impact analysis[C]// IEEE Computer Society, 1996:1-25.

[3] Unified Modeling Language : Infrastructure version 2.0.[EB/OL]. http://www.omg.org/technology/documents/formal/uml.htm

[4] IEEE Standard Glossary of Software Engineering Terminology[S]. IEEE Std 610.12-1990, Published by the Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, 1990.

[5] Haney F M. Module connection analysis-a tool for scheduling of software debugging activities[C]// In Proceedings of AFIPS Fall Joint Computer Conf, 1972:173-179.

[6] Ren X, Ophelia C. Identifying failure causes in Java programs: an application of change impact analysis [J]. IEEE Transactions on Software Engineering, 2006,32(9): 718-732.

[7] XUI RIA Framework [EB/OL]. http://sourceforge.net/projects/xui/