一种基于结构查询的UML设计模式识别方法

许涵斌[1] 张学林[1] 郑晓梅[2] 张 天[1] 李宣东[1]
(南京大学计算机软件新技术国家重点实验室 南京 210023)[1]
(南京中医药大学信息技术学院 南京 210023)[2]

摘 要 随着模型驱动技术的逐渐成熟和广泛应用,大量反映程序结构、行为以及性质的模型产生于软件的开发过程中,并成为软件文档的重要部分保存下来。其中,尤以UML模型的应用最为广泛,也因此形成了通过理解UML模型来理解大规模、高复杂性软件系统的研究思路。对UML模型理解的一个难点是如何有效地从大量复杂的模型中,快速查找并定位具有一定结构特征的模型片段。幸运的是,设计模式的普遍应用为我们快速、高效地理解和定位模型提供了一条重要的线索。然而,随着技术的发展,设计模式数量在不断增长,其自身在应用中的结构也在微妙变化,这些都给相应查询和识别工具的开发带来一定的困难。文中从查询和匹配UML模型中特定结构的角度入手,利用UML自身特点设计相应查询算法,通过分析和理解设计模式的结构特征,从UML模型中查询相应的设计模式,以达到灵活和高效地理解软件系统的目的。

关键词 模型查询技术,统一建模语言,信息抽取,设计模式
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2014.11.011

UML Design Pattern Recognition Method Based on Structured Query

XU Han-bin[1] ZHANG Xue-lin[1] ZHENG Xiao-mei[2] ZHANG Tian[1] LI Xuan-dong[1]
(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China)[1]
(School of Information Technology, Nanjing University of Chinese Medicine, Nanjing 210023, China)[2]

Abstract As model-driven techniques are matured and widely used, more and more models reflecting structures, behaviors and features of program have been produced in the process of software development. And models also preserve as important parts of software documentations. Among them, UML models are most widely used. Therefore, comprehension of UML models is thought as a good way to the comprehension of large-scale, highly complex software systems. One of the difficulties to comprehension of UML models is how to find and locate effectively a certain structural feature of model fragments from a large number of complex models. Fortunately, the wide application of design patterns provides an important clue for us to understand and locate model quickly and efficiently. This paper aimed to analyze and understand the structural features of design patterns in order to identify design patterns in UML models. In this way, the purpose of understanding software system flexibly and efficiently can be achieved.

Keywords Model query technology, Unified modeling language, Information extraction, Design patterns

1 引言

随着开源浪潮的推进,我们手上拥有了大量可复用的开源工程的代码。但是,这些开源工程的代码往往十分繁琐和冗长,不便于直接理解和复用。因而如何高效灵活地理解代码,以提高复用效率已经成为我们亟需解决的问题。幸运的是现代软件生产中设计模式得到了广泛的使用,这为我们以设计模式为线索理解软件、复用代码提供了可能。

设计模式(Design pattern)[8]为面向对象的软件开发提供了可重用的设计方案,有利于提高软件的复用性和系统的可维护性,在工业软件开发中获得了广泛的应用。

我们在对一个复杂的软件系统进行理解时,以设计模式为线索,往往可以快速掌握主要模块的组织结构,有时甚至能够了解到设计人员对系统的设计思路和理念。

统一建模语言(Unified Modeling Language, UML)[2]是一种通用的可视化建模语言,目前已经被普遍应用于各种软件的设计和开发中。随着模型驱动工程[15]的日益成熟和广泛应用,通过UML构造的系统模型已经逐渐成为系统文档的主要组成部分,并随系统的可执行程序一起进行演化。因此,通过UML模型来理解系统,已经成为软件开发和维护人

到稿日期:2013-09-16 返修日期:2013-11-16 本文受国家自然科学基金(61003025,61021062),国家863高技术研究发展计划(2011AA010103,2012AA011205)资助。

许涵斌(1991—),男,硕士生,主要研究领域为模型驱动工程,E-mail:xuhanbin@seg.nju.edu.cn;**张学林**(1989—),硕士生,主要研究领域为模型驱动工程;**郑晓梅**(1979—),女,讲师,主要研究领域为软件工程;**张 天**(1978—),男,副教授,主要研究领域为模型驱动工程,E-mail:ztluck@nju.edu.cn(通信作者);**李宣东**(1963—),男,教授,主要研究领域为模型检验、软件工程。

员由抽象到细节理解系统的一个重要环节。

通过理解 UML 模型中的设计模式，可以更加快速直观地理解软件系统。然而，由于设计模式的数量随着软件开发技术的延伸而不断增加，使得针对特定设计模式结构的识别有很大的局限性。此外，即使是同一种设计模式，由于应用场景不同，其结构也会发生变化，这就更加增大了对于设计模式识别工具的开发难度。

针对设计模式的识别往往是基于其结构特征的，如果可以将此类结构特征的定义从识别过程中剥离出来，形成一个独立的环节，也就可以灵活地应对和处理各种新的设计模式或者模式的变型。

通过对此类问题的研究，我们发现，针对 UML 模型中设计模式的识别，可以看作是针对图(graph)数据结构的子图匹配问题。识别出一种新的设计模式，就相当于去匹配一种新的子图结构，其过程在本质上是一样的。这个问题其实就是数学上的子图同构问题(Subgraph isomorphism problem)[3]。

我们在之前所提出的基于结构匹配的模型查询技术的基础上，针对 UML 模型自身的特点，进一步改进了查询算法，使之可以更好地应用于大规模模型的查询和识别，并针对主流的 23 中设计模式进行了实例应用。

本文第 2 节介绍一种经典的基于结构匹配的模型查询技术[7]；第 3 节针对 UML 模型自身的特点，对基于结构匹配的模型查询技术进行优化和改进；第 4 节详细介绍在 UML 模型中对设计模式进行识别的具体方法；第 5 节进行相关工作比较；最后总结全文并对下一步工作进行展望。

2 基于结构匹配的模型查询技术

本文所提出的 UML 设计模式识别方法，是对作者之前所提出的一种基于结构匹配的模型查询技术[7]加以扩展得到的。

2.1 UML 模型文件的结构信息抽取

由于 UML 模型保存得到的 XMI 格式文本文件满足 OMG 所制定的文档类型定义(Document Type Definition，DTD)[4]，XMI 格式文本文件具有极强的结构特征，在遍历 XMI 格式文本文件过程中可以根据 XMI 的结构特征规律进行 UML 模型元素的信息抽取，将 XMI 中的模型结构抽取成一个个不同类型的模型元素节点，为我们下一步模型结构的匹配提供了条件。

基于结构匹配的模型查询技术使用深度优先的算法[6]来抽取 UML 模型文件的结构信息(见算法 1)，本文使用以 SAX(Simple API for XML)[5,9]为基础的 XML 文件解析工具对 XML 文件进行解析，这种 XML 文件解析工具是一种基于事件的解析工具。针对不同的模型元素在 XMI 文件中的文本结构，我们可以设置不同的事件处理函数，即对不同的模型元素编写相对应的 startElement()和 endElement()函数来实现对模型结构信息的抽取，并将抽取出的结构信息以信息节点的方式存储在内存中，以供后续操作的调用和处理。其中对于类元类型的模型元素，我们设置的事件处理函数首先要抽取类元本身的一些属性值如 id、name、visibility、isAbstract 等；其次针对类元的事件处理函数还要包含对类元的 Attribute 和 Operation 的属性值的抽取方法。针对类间关系的事件处理函数则相对来说更简单一些，只需要抽取类间关系的 ID 和类间关系两端类元的 ID 即可。

算法 1 结构信息的抽取

```
输入：模型的根节点 root
输出：抽取出的信息节点
VISITELEMENT(root)
1. FOR every child of root do
2.    startElement();Δ 开始访问当前节点
3. VISITELEMENT(child);Δ 访问子节点
4.    endElement();Δ 结束访问当前节点
```

2.2 UML 模型匹配

在对 UML 模型文件的结构信息完成抽取后，我们得到了一系列 UML 模型的结构信息节点，根据这些节点，我们便可以进行 UML 模型的匹配。UML 模型的匹配主要分为 3 部分：第一部分为类元的匹配算法；第二部分为类间关系的匹配算法；第三部分为整个模型的匹配算法。

2.2.1 类元的匹配

类元的匹配首先是对类元的一些属性值进行匹配如 name、visibility、isAbstract 等，name 可以使用字符串比较的方式进行匹配，visibility 是枚举类型，其他的特征值是 boolean 类型，这些都可以直接进行比较，然后如果类元中存在类元的属性 Attribute 和类元的方法 Operation，那么还要对类元的 Attribute 和 Operation 进行比较匹配，如果以上这些全都匹配，我们才能认定这两个类元匹配。

算法 2 类元的匹配算法

```
输入：用户类元 userclass，模型库类元 patternclass
输出：匹配标志
MatchClass(userclass,patternclass)
1. IF Property values between userclass and
2.    patternclass is not match Δ 匹配属性值
3.    THEN return false;
4. IF !MatchAttribute(userclass,patternclass)
5.       || !MatchOperation(userclass,
6.       patternclass)
7. THEN return false;Δ 匹配 Attribute 和 Operation
8. return true
```

2.2.2 类间关系的匹配

由于类间的关系的文本文件的结构比较简单，因此在匹配过程中只要比较类间关系的两端元素是否匹配就可以判断此类间关系是否与模型库中的类间关系相匹配。同时对类间关系进行匹配的过程中，在查找类间关系时，存在两种情况：关系一端的类元元素已经与模型库中的类元匹配和关系两端的类元元素都已经与模型库中的类元匹配。这两种情况中若两端元素都已经匹配，便可以认为类间关系已经匹配；若只有关系一端元素匹配，则还要探查另一端的类元；若另一端的类元不与模型库中的类元相匹配，则该关系不匹配，需要回退寻找其他匹配的类间关系。

由于类间关系的结构和语法相似，以下仅以依赖关系为例说明类间关系的匹配算法。

算法 3 类间关系的匹配算法

```
输入：类元节点 element，模型库 modelbase
输出：类间关系的匹配映射 matchResult
Dependencymatch(element)
1. IF two sides of element have been visited
2.    THENIF element exists in modelbase
```

```
3.      match(matchResult,matchedIndex+1);
4.    ELSE
5.        return;Δ没有对应匹配的关系
6. ELSE IF only one side of element has been visited
7.    THEN IF element exists in modelbase
8.    THEN add mapping between two IDs;
9.        match(matchResult,matchedIndex+1);
10.         remove mapping between two IDs;
11.     ELSE
12.       return; Δ没有找到匹配的关系
```

2.2.3 整个模型的匹配

本文使用深度优先的匹配算法对整个模型进行匹配。首先使用深度优先的遍历方法遍历整个待匹配的模型，生成一个用于匹配模型节点序列；按照序列的顺序进行模型元素的匹配，匹配方法使用 2.2.1 节和 2.2.2 节中的类元匹配方法和类间关系匹配方法，若模型元素在模型库中找到匹配元素，则建立匹配元素之间的 ID 映射，同时深度加 1，进行下一个模型元素的匹配，直到深度等于新设计模型的元素个数表明已完成匹配，输出匹配的节点 ID 映射；若模型元素在模型库中找不到匹配元素则回退到上一层，同时将之前建立的节点映射删除，若回退到第一层且遍历完第一层所有找到的匹配元素仍未找到匹配的模型映射，则匹配失败。

算法 4 整个模型的匹配算法

```
输入:模型元素映射表 matchResult,匹配模型的深度 matchedIndex,
     待匹配模型 model,模型库 modelbase
输出:模型元素的匹配映射表 matchResult
match(matchResult,matchedIndex)
1. IF matched Index equals model size
2. THEN print matchResult;Δ输出节点映射关系
3.      return;
4. FOR each element in model Δ模型元素逐个匹配
5.   do IF element is Classifier Δ类元元素
6.     THEN IF element has not been visited
7.       THEN FOR each matched element in model
8.         do add mapping between two IDs;
9.         match(match Result,matchedIndex+1);
10.       remove mapping between two IDs;
11.   ELSE FOR each matched element in model
12.     do match(matchResult,matchedIndex+1);
13.       remove mapping between two IDs;
14. ELSE IF element is Abstraction Δ实现关系
15.       Then Abstraction-match(element);
16. ELSE IF element is Dependency Δ依赖关系
17.       THEN Dependencymatch(element);
18. ELSE IF element is Generalization Δ继承关系
19.       THEN Generalizationmatch(element);
20. ELSE IF element is Association Δ关联关系
21.       THEN Associationmatch(element);
```

3 模型的匹配算法改进

经典的基于结构匹配的模型查询技术是单纯地从数学的角度将模型的匹配转变成为子图的匹配，没有充分考虑 UML 语言所拥有的语义特点，因而我们可以针对 UML 图这一特定领域进行模型匹配算法的优化，提高匹配算法的整体效率。

3.1 UML 模型元素分析

从 UML 图中抽取出来的节点包含有大量的 UML 的结构信息，通过对 UML 语义进行研究，我们可以找到 UML 图中模型元素的一些规律，根据这些规律可以对匹配算法做定向的优化。

为了找出 UML 图中模型元素的一些规律，我们对一些开源软件工程的模型元素和模型元素各个属性的属性值进行了统计，其中用 Java 编写的 BitTorrent 客户端 VUZE 共拥有 9MB 的源代码，代码量较大，得出的结果比较具有代表性，其中模型元素出现频率的统计结果如表 1 所列。

表 1 模型元素出现频率表

模型元素类型	出现频率
Classifier	11%
Attribute	26%
Operation	49%
Relationship	14%

由于模型元素的属性值比较繁多，以下仅列出 Classifier 模型元素的属性值分布表，并通过此表来说明模型元素的属性值的分布特点。

表 2 Classifier 模型元素的属性值分布表

属性	值	数目	频率
name	*	1	0.0306%
visibility	public	1169	35.74%
	private	46	1.41%
	protected	129	3.94%
	package	1927	58.91%
isAbstract	true	50	1.53%
	false	3221	98.47%

通过对模型元素和模型元素的各个属性的属性值进行统计可以发现，在 UML 模型中各种不同的模型元素和模型元素各个属性的属性值出现的频率是不同的，而且出现频率的差别十分的巨大，例如 Attribute 模型元素出现的频率只有 Operation 模型元素出现的频率的一半，对于类元模型元素中枚举类型的属性 isAbstract，不同的值出现的频率差别更大，在统计的 3271 个类元中只有 50 个类元的 isAbstract 属性为 true，其余的都为 false，true 只占总数的 1.5%。

3.2 基于 UML 的匹配算法改进

进行模型匹配的过程实际上就是在模型库中排除子图的过程，比如说在寻找匹配第一个节点时有 m 个匹配的节点，由此产生了拥有 m 张相符的子图的集合，在选取与第一个节点相连的边的匹配过程中由于只有部分的边相符因此减少了 s 张符合的子图，使得子图的集合缩小，如此进行下去直到找到匹配子图为止。如果一开始我们就选取比较稀有的节点进行匹配，那么一开始所产生的匹配的子图集合的规模就小了许多，在之后进行匹配的过程中还可以不断优先选取稀有的节点和边进行匹配，这可以不断地缩小子图集合的匹配规模，直到找到匹配的子图为止。因而输入队列对于匹配结果的影响很大。

通过 3.1 节的分析，我们可以知道受到 UML 语义的影响，UML 模型元素和模型元素各个属性的属性值出现的频率有很大的差别，这为我们根据 UML 模型元素和模型元素的各个属性的属性值出现规律进行匹配算法的定向优化提供了可能。

针对匹配算法的特征和 UML 模型元素的分布情况，我

们可以采用改变预处理后的输入序列排序的方法来减少匹配算法的搜索匹配次数。本文通过对 UML 模型元素及其属性值进行优先级划分，并在对 UML 模型元素节点进行预处理时采取优先级低的先进栈优先级高的后进栈的策略，使得产生的预处理输入序列优先级高的在序列的前端，优先级低的在序列后端。这使得在进行模型元素的搜索匹配时优先级高的被先访问，优先进行模型元素的匹配。如果优先级高的模型元素存在匹配，就可以大大减少递归搜索探查的次数，预处理生成序列的算法如下。

算法 5 输入序列生成算法

```
输入：待匹配模型 queryPattern
输出：预处理序列 list
sortedQueryPattern(query Pattern)
1. Push(firstClass)；Δ将优先级最高类元进栈
2. WHILE queryPattern or stack is not empty
3.   doelement←Pop()；Δ元素逐个出栈
4.     list[i++]←element；
5.     IF element is Classifier
6.       THEN push all relationship
7.              connected element into
8.              stack by Priority；
9.     ELSE IF element is Dependency
10.      THEN push all classifier
11.             connected element into
12.             stack by Priority；
13.    ELSE IF element is Abstraction
14.      THEN push all classifier
15.             connected element into
16.             stack by Priority；
17.    ELSE IF element is Generalization
18.      THEN push all classifier
19.             connected element into
20.             stack by Priority；
21.    ELSE IF element is Association
22.      THEN push all classifier
23.             connected element into
24.             stack by Priority；
```

3.3 改进的匹配算法分析

改进的匹配算法同原匹配算法一样都是 np 完全性问题，因此我们需要假定一些条件来简化 np 完全性问题，以实现对改进的匹配算法的时间复杂度分析。

我们假定要匹配的模型是一条具有 k 个节点的路径，定义稀有度为该节点在模型库中可以找到的匹配节点数，k 个节点的稀有度分别为 x_i，假设模型库与该路径中的节点类型相同的节点共有 n 个，$x_i \leqslant n$。

首先我们将路径节点按照稀有度来排序遍历，可以得到一个按节点稀有度递增的遍历序列，序列中第 i 个节点的稀有度为 y_i。因而在最差情况下，如果路径在模型库中存在，那么在模型库中寻找路径的第一个节点要与模型库中的节点比较 n 次，在模型库中寻找路径的第二个节点要与模型库中的节点比较 $n * y_1$ 次，在模型库中寻找路径的第 s 个节点要与模型库中的节点比较 $n * \prod_{1}^{s-1} y_i$ 次，整个路径的遍历过程中总计要与模型库中的节点比较 $n * (1+\sum_{1}^{k-1}\prod_{1}^{j} y_i)$ 次。如果使用未改进的匹配算法进行匹配，那么遍历过程中查找比较次数为 $n * (1+\sum_{1}^{k-1}\prod_{1}^{j} x_i)$，由于 $\{y_i\}$ 序列是 $\{x_i\}$ 序列的递增排序，因此 $y_1 * y_2 * \cdots * y_s \leqslant x_1 * x_2 * \cdots * x_s$ 总是成立的，因而路径上每一个节点的搜索次数都比改进前要少，因此搜索匹配的效率比改进前高。改进算法本质上是一种贪心算法。

由于设计模式的结点个数通常比较少，一般设计模式的模型图中的节点不会超过 7 个，因而我们对 7 个节点以内的路径进行实验便可以有效地对改进算法的效率进行评估。

在实际的运行环境中，对 4 个、5 个、6 个、7 个节点的路径分别进行匹配实验，同时设置数据库的规模从 40 个模型元素逐步增加到 1000 个模型元素，以此来验证匹配模型和数据库在不同的数据规模下算法改进前后效率的对比。为了展示算法改进前后效率的对比，我们选取 6 个节点的路径在数据库中的匹配作为典型的实验案例。

在 6 个节点的路径匹配实验中，我们设计了能在模型库中找到匹配模型和未能在模型库中找到匹配模型这两组实验来验证我们的改进算法的有效性，其实验数据的柱状对比如图 1、图 2 所示。

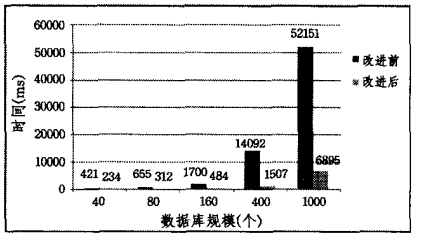

图 1 未找到匹配模型消耗时间图

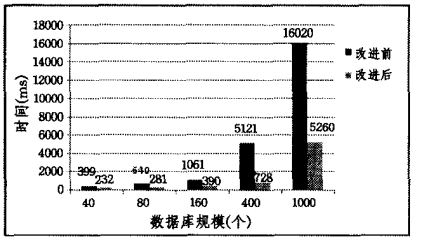

图 2 找到匹配模型消耗时间图

从图中可以发现，在数据库的规模从 40 个模型元素逐渐增加到 1000 个模型元素的过程中，改进后的匹配算法运行所花费的时间总是小于改进前，数据库规模越庞大，改进算法较原算法的效率提高就越明显，当数据库规模达到 10^3 数量级时，效率平均可以提高 5 倍到 10 倍。如果在模型数据库中不存在要匹配的模型，改进算法花费的时间更是远远少于原算法。由此我们可以得知改进算法是切实有效的、可行的。

4 设计模式的识别

设计模式被广泛应用于现代软件工程的方方面面，为软件的开发提供了一套比较完善的系统方案。而且由于设计模式大致可以分为创建型的设计模式、行为驱动的设计模式、结构驱动的设计模式、语言相关的设计模式、特定领域相关的设计模式这几类，其类别比较有限，因此我们可以对这些设计模式的模型结构图进行识别，从而达到灵活高效地理解软件的目的。

4.1 设计模式模型图的获取

在设计模式的实际使用中，创建型的设计模式、行为驱动的设计模式、结构驱动的设计模式使用得最为广泛和频繁，而

且这几种类型的设计模式的模型结构也十分清晰。我们可以使用 UML 类图的绘制工具将这些设计模式的模型图绘制出来，在之后的设计模式识别的过程中作为待匹配的模型图使用。以下仅以桥梁模式为例，说明设计模式模型图的一些特征[8]。

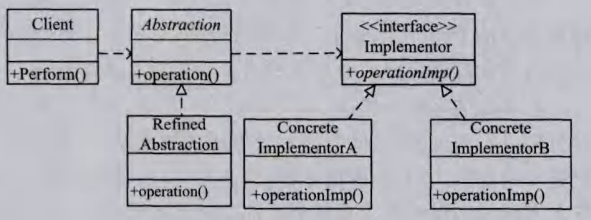

图 3　桥梁模式模型图

图中类元模型 Abstraction 是对给出的定义进行抽象化处理；类元模型 Refined Abstraction 是对已经抽象化的角色进行扩展，对父类进行修改和修正；类元模型 Implementor 则是现实化角色的接口；而类元模型 Concrete Implementor 则是给出现实化角色接口的具体实现。类元模型元素之间的关系如下：Client 和 Abstraction 之间是依赖关系，Abstraction 和 Refined Abstraction 之间是继承关系，Abstraction 和 Implementor 之间是聚合/组合关系，Implementor 和 Concrete Implementor 是实现关系。

在对软件进行理解时，只要识别以上的结构，即可以说明软件在设计时有一部分的系统结构使用了桥梁模式这种结构驱动的设计模式。

4.2　设计模式的识别

要对设计模式进行识别首先必须要建立一个完备的模型库，本文使用开源的软件项目 VUZE 作为模型库。在建立模型库时，对于那些拥有完整的设计模式模型图如 UML 类图等静态图的开源项目，我们将其设计模式的模型图直接作为模型库使用；对于那些只有代码、没有相应的 UML 设计模式模型图的开源软件项目，我们可以使用一些转换工具如 ArgoUML 让代码自动逆向生成其对应的设计模式的模型图作为模型库使用。在匹配时，我们将把模型库中的元素信息抽取出来和用户模型进行比较匹配。

由于我们需要在实际的项目中识别出用户指定的设计模式，因此在用户需要识别的设计模式模型中不能带有实际的信息如类名、关系名等，也就是说用户输入的设计模式模型图只能是设计模式的抽象结构图。因此我们首先要对设计模式的 UML 类图做模糊化处理，将其中一些具体的信息如类元的名称、关系的名称等去除，留下一个设计模式的 UML 类图的结构框架，然后对这个结构框架进行结构信息的抽取，将信息保存为一个个的节点，再根据事先设定好的优先级对节点进行排序处理得到一个匹配序列与模型库中的模型进行匹配，其流程如图 4 所示。

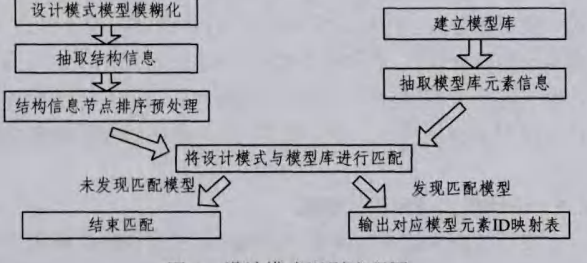

图 4　设计模式识别流程图

4.3　实例分析

本文对主流的 23 种设计模式均作了验证，23 种设计模式均可以被识别。本节将使用较为简单的适配器设计模式作为典型来详细地叙述如何使用本文提出的设计模式识别方法进行设计模式的识别。

首先，假定用户需要在开源项目中对适配器模式进行设计模式的识别，要识别的适配器模式大致可以用图 4 所示的适配器模式的 UML 类图文件来表示。在获得用于进行设计的模式识别后，第一步是对模型进行模糊化处理，将其中的具体的信息如类名、类间关系名等去除后，得到了一个类似图 5 的模糊处理过后的只有框架结构信息的模型。

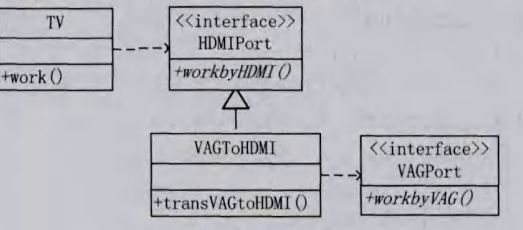

图 5　适配器模式模型图

对用户设计模式模型进行模糊化处理后，第二步是使用 SAX 的 UML 解析工具对适配器设计模式的 UML 类图进行结构化的信息抽取，并将模型的结构信息以节点的方式存放在计算机内存中，并将这些信息节点按优先级（I1＞C1＞C2＞I2）排序，得到一个预处理的输入序列，其顺序为 $$02→$$05→$$01→$$06→$$03→$$07→$$04。适配器模式的结构如图 6 所示。

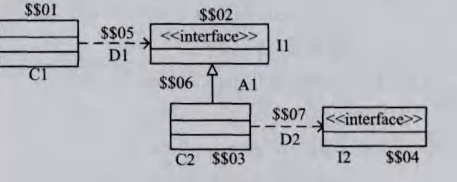

图 6　适配器模式的结构图

在得到预处理的输入序列后，第三步便是按照输入序列的顺序，对模型元素进行逐个的匹配，若全部匹配成功则输出相应节点的 ID 映射（见图 7），由于模型元素 ID 的唯一性，我们便可以根据模型 ID 定位到相应的模型，实现对软件的设计模式的识别。若没有全部匹配成功，则表明软件中不存在相应的设计模式。

```
Mapping Result:
$$01->-84-26-16--102-1889f578:13eb645fd22:-8000:000000000000B62
$$02->-84-26-16--102-1889f578:13eb645fd22:-8000:000000000000B67
$$03->-84-26-16--102-1889f578:13eb645fd22:-8000:000000000000B6A
$$04->-84-26-16--102-1889f578:13eb645fd22:-8000:000000000000B6F
$$05->-84-26-16--102-1889f578:13eb645fd22:-8000:000000000000B72
$$06->-84-26-16--102-1889f578:13eb645fd22:-8000:000000000000B6B
$$07->-84-26-16--102-1889f578:13eb645fd22:-8000:000000000000B73
```

图 7　匹配节点的 ID 映射图

5　相关工作

传统的设计模式识别方法通常采用相似度匹配[10]、模板匹配[10]、DNIT（Depth-Node-Input Table）[11]匹配以及利用控制流（CFG）图进行匹配[12,13]等方法，这些方法有的将设计模式的模型图转换为矩阵，通过计算矩阵的相似度的方法对设计模式进行识别，有的通过改变匹配图中的节点来计算匹配图的相似性从而实现设计模式的识别，还有的利用基于基本块的数据流分析方法进行设计模式的识别。

相似度匹配[10]直接利用图的矩阵表示法对图的节点进行相似度计算，其缺陷在于这种设计模式的识别方法只注重各个节点之间的相似程度，而忽略了整张模型图的相似程度，这样只能保证各个节点的匹配而不能够保证整幅模型图的匹配，因而准确度不够理想。

模板匹配[10]则是根据不同的关系来构造多个关系矩阵，然后关系矩阵利用特征值合并归一的方式得到一个特征矩阵，进而对两个图的特征矩阵使用例如 NCC 算法（Normal Cross Correlation）[14]等相似性计算算法来计算两个图的相似性，然而这种方式虽然准确度较相似度匹配高，但是它仍然注重于节点相似而不是全图相似，只是其携带的结构信息更多。

DNIT（Depth-Node-Input Table）[11]匹配也是一种图匹配算法，它将图匹配的过程分解成为 k 步，k 的值为两幅图中的一幅经过变换后若干个节点可以变得相同的节点个数。首先利用一个由祖先节点的数目、子女节点的数目、兄弟节点的数目组成的三元组（t1，t2，t3）来表示模型图中的一个节点，然后再根据三元组使用一些算法来计算出每一个节点的特征值，这样便将整张模型图的节点信息转换成了该图的节点特征值表，然后再计算设计模式的模型图和用户系统模型图对应的特征值表之间的距离，得到一个距离矩阵 P，然后使用迭代的方法迭代 K 步，每次迭代的过程都找出 P 中每行的最小值，最终得到两幅图的相似度，借此来实现设计模式的识别。

利用控制流（CFG）图进行设计模式识别[12,13]的方法是利用分析数据流的方式来分析系统的行为模式，得到一个个的基本块，将这些分析得到的基本块串联起来便得到了控制流图，利用控制流图对设计模式进行识别。

传统的设计模式识别的方法计算矩阵相似度过程非常繁琐，对于设计模式的识别准确度也不高，而且将设计模式的模型转化成数字矩阵难以给人一个直观的识别过程的认识。文本提出的这种基于结构驱动的模型查询技术的设计模式的识别方法很好地克服了以上这些问题，识别过程清晰明了，识别的准确性也比较高。

结束语 本文基于之前提出的 UML 模型查询技术，进一步提出了一种基于结构查询的 UML 设计模式识别方法。针对之前模型查询技术中匹配算法由于使用递归而导致时间开销过大的问题，进行了基于 UML 的定向优化改进，有效提高了查询的效率。本文提出的针对 UML 的设计模式识别方法，能够灵活有效地查询 UML 模型中的特定结构，从而识别出相应的设计模式。此外，由于本文的设计模式结构特征是通过自定义的方式给出的，因此它具有良好的可扩展性。

下一步，我们还可以尝试对算法进行进一步的优化，使算法的运行效率更高，同时我们还打算将该工作完全移植到 EMF 框架下。

参 考 文 献

[1] Liu Hai-yan, Liang Jian-long, Suo Zhi-hai, et al. Design pattern and their applications to software design[J]. Journal of Xi'an Jiaotong University, 2005, 39(10): 1043-1047

[2] Lu Bo, Chai Yue-ting. On Unified Modeling Language—UML [J]. Computer Engineering and Science, 2000, 22(4): 58-60

[3] Subgraph Isomorphism Problem [EB/OL]. [2013-07-09]. http://en.wikipedia.org/wiki/Subgraph_isomorphism_problem

[4] Document Object Model (DOM) [EB/OL]. [2009-01-06]. http://www.w3.org/DOM/

[5] Wang Fang, Li Zheng-fan. The Realization Method of Parsing XML Document by SAX[J]. Journal of East China Jiaotong University, 2004, 21(1): 84-86

[6] Cormen T H, Leiserson C E, Rivest R L, et al. Introduction to Algorithms(Second Edition)[M]. BeiJing, China: China Machine Press, 2006

[7] Zhang Xue-lin, Zhang Tian, Li Xuan-dong. Query by Drawing Examples of UML Model[C]// Proceedings of the Software Engineering Conference (APSEC), 2012 19th Asia-Pacific. Hongkong, China, 2012: 154-157

[8] Gamma E, Helm R, Johnson R, et al. Design Patterns: Elements of Reusable Object-Oriented Software[M]. BeiJing, China: China Machine Press, 1995

[9] Java API for XML Processing (JAXP) Tutorial [EB/OL]. [2008-07]. http://www.oracle.com/technetwork/java/sax-138988.html

[10] Dong Jing, Sun Yong-tao, Zhao Ya-jing. Design pattern detection by template matching[C]// ACM Symposium on Applied Computing-SAC. 2008: 765-769

[11] Pandel A, Gupta M, Tripathi A K. DNIT—A new approach for design pattern detection[C]// International Conference on Computer and Communication Technology-ICCCT. 2010

[12] Gupta M, Rao R S, Tripathi A K. Design Pattern Detection using inexact graph matching[C]// Proceedings of the International Conference on Communication and Computational Intelligence. 2010: 211-217

[13] Shi Ni-ja, Olsson R A. Reverse Engineering of Design Patterns fromJava Source Code[C]// Automated Software Engineering-ASE. 2006: 123-134

[14] Lewis J P. Fast Template Matching[J]. Vision Interface, 1995, 5: 120-123

[15] Schmidt D C. Guest Editor's Introduction: Model-Driven Engineering[J]. IEEE Computer, IEEE CS, 2006, 39: 25-31

（上接第 24 页）

[2] Chandra A, Chakrabarty K. Frequency-directed Run-length (FDR) Codes with application to system-on-a-chip test data compression[C]// Proceedings of IEEE VLSI Test Symposium, 2001. Washington, DC, USA: IEEE Computer Society, 2001: 42-47

[3] Chandra A, Chakrabarty K. Reduction of SOC Test Data Volume, Scan Power and Testing Time Using Alternating Run Length Codes [C] // Design Automation Conference, 2002. Washington, DC, USA: IEEE Computer Society, 2002: 673-678

[4] Touba N A. Survey of Test Vector Compression techniques[J]. IEEE Design & Test of Computers, 2006, 23(4): 294-303

[5] 梁华国，蒋翠云. 基于交替与连续长度码的有效测试数据压缩和解压[J]. 计算机学报，2004，27(4)：548-554

[6] 韩建华，詹文法，查怀志. 一种相对游程长度编码方案[J]. 计算机科学，2012，39(5)：295-299