Cross-application Fan-in Analysis for Finding Application-specific Concerns

Makoto Ichii[†] Takashi Ishio[†] Katsuro Inoue[†]

[†]Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
E-mail: {m-itii, ishio, inoue}@ist.osaka-u.ac.jp

Abstract

Automatic detection of crosscutting concerns helps comprehension and refactoring of large-scale software system. Some of the detection techniques use occurrence frequency of program entities to detect concerns; however, the detected concerns include generic idioms such as a loop using iterator. The generic idioms are less interesting than application- or domain-specific concerns in many cases. Therefore, some techniques use heuristics to filter the concerns, where developers should adjust the filter carefully or actual concerns may be accidentally filtered out. In this paper, we propose a metric named universality that represents how widely a class is used in applications. Using the universality, we filter concerns comprising only universally-used classes. We apply the method to coding patterns that include crosscutting concerns in various open source applications and discuss the result.

Keywords:Crosscutting concern, Coding pattern, Fan-in analysis

1 Introduction

Crosscutting concern detection techniques help developers to refactor or comprehend the target software system. There are some techniques automatically detecting crosscutting concerns based on their nature: their implementation appears across the source code [1, 5, 6]. Coding pattern detection [5] is one of such methods using a data mining technique. A coding pattern consists of frequent sequence of method calls with control statements.

However, detected patterns include some generic patterns (idioms) such as the iterator idiom in addition to the application- or domain-specific concerns[1]. The generic patterns are less helpful for developers because they are not interested in already-known patterns. This matter is not unique to the coding pattern detection technique, but common in almost all of the methods based on the occurrence frequency of concerns. Marin et al. [6] excludes all libraries from target application as default settings of their fan-in analysis in order to exclude generic utilities; however, such filtering accidentally removes concerns with library classes.

In this paper, we propose cross-application fan-in analysis to find classes universally used across various applications/domains. Coding patterns comprising only universally-used classes can be filtered out as generic patterns. We define *universality* metric for a class that how widely a class is used. We have computed the universality metric values for a collection of open-source software, and applied the metric to filtering coding patterns that comprise only universal classes, such as loops using iterator.

This paper is constructed as follows: Section 2 describes the related works on detection of the crosscutting concerns as the background. Then we explain our analysis method in Section3 and its case study in Section 4. We conclude this paper in Section 5.

2 Background

The crosscutting concern detection (or the aspect mining) techniques help developers to manage idiomatic code fragments. Fung [5] is a tool to detect coding patterns including crosscutting concerns from an application. A coding pattern detected by Fung consists of an ordered list of method calls and control statements. Figure1 is an example. The left code snippet and the right box are a Java implementation of a loop using an iterator and its coding pattern representation, respectively. Marin et al. proposes fan-in analysis method to extract method calls that consists crosscutting concerns [6]. Bruntink et al. applies code clone detection for finding crosscutting concerns [1]. However, frequent code fragments include not only crosscutting concerns but well-known implementation idioms such as a loop with an iterator since the idioms are also frequently used in programs. To filter out implementation idioms, Marin's fan-in analysis excludes library classes from the result. Config-

[1]In this paper, "application" means a software system; "domain" means a group of applications having a common horizontal/vertical feature.

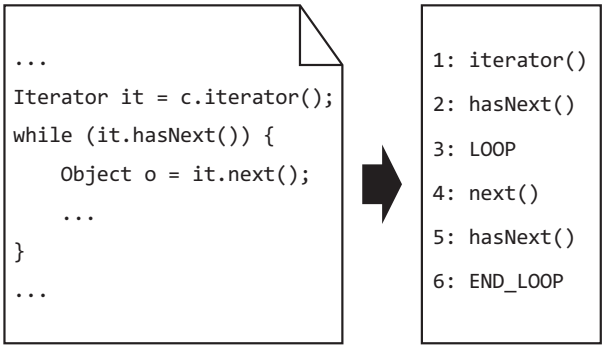

Figure 1. Coding pattern example

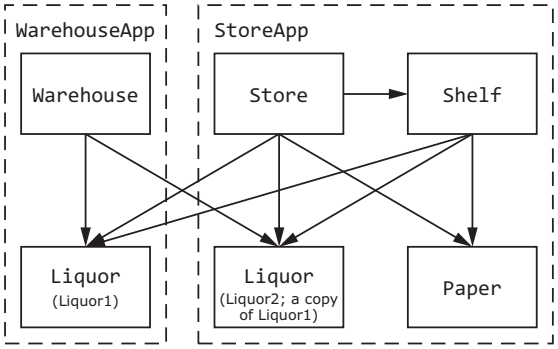

Figure 2. Component graph example

uring a library filter requires developers to distinguish application classes from library classes for each application. Without an appropriate configuration, a library filter may accidentally remove interesting code fragments.

3 Cross-application Fan-in Analysis

In this paper, we propose cross-application fan-in analysis in order to automatically distinguish implementation idioms from coding patterns. The key idea is that implementation idioms appear in various applications, while application- and domain-specific coding patterns are involved in only a particular set of applications. We propose a metric named *universality* to measure how widely a class is used across applications. We use a collection of applications to compute universality metric for each class and apply the metric to extract crosscutting concerns in a target application.

3.1 Cross-application use-relation

First of all, an application collection is set up so that the application collection contains the target application, the libraries used by the application and other applications using the libraries.

Then, static use-relation between the classes in the application collection is analyzed.

We construct a use-relation graph, or a directed graph whose node represents a class and edge represents use-relation between classes respectively. An edge represents one or more of the following relation [4] :

- A class or an interface *extends* another class or interface respectively.
- A class *implements* an interface.
- A class or an interface *declares a variable* (a local variable, a field, a parameter, or the return type of a method) *of* a class or an interface.
- A class *instantiates* a class object.
- A class *calls a method of* a class or an interface. If the callee method is inherited from a parent class (or interface) of the callee class (or interface), we interpret that the method of the parent class (or interface) is called.
- A class or an interface *refers to a field of* a class or an interface. If the referenced field is inherited from a parent class (or interface) of the referenced class (or interface), we interpret that the field of the parent class (or interface) is referred to.

If there are multiple copies of a class in different applications, we employ all of the classes as used classes. Figure2 is an example of the use-relation graph that consists of six classes from two applications (`WarehouseApp` and `StoreApp`). `WarehouseApp` contains two classes, where `Warehouse` uses `Luquor` (Liquor1). `StoreApp` comprises four classes: `Store`, `Shelf`, `Paper` and `Liquor` (Liquor2) that is a copy of Liquor1. In `StoreApp`, `Store` uses `Shelf`, `Luquor` (Liquor2) and `Paper`; `Shelf` uses `Liquor` (Liquor2) and `Paper`. Because Liquor2 is a copy of Liquor1, `Warehouse`, `Store` and `Shelf`, the classes containing references to "Liquor", have use-relations to both of the two `Liquor` classes.

3.2 Class universality

The universality metric is measured using the constructed use-relation graph. The universality uses following two kind of fan-in: the *class fan-in* and the *application fan-in*. The class fan-in of a class c is the number of classes using the class c, that is, the number of incoming edges of the node corresponding to the class c. For example, both of the `Liquor` classes of Figure 2 have the value of 3. The application fan-in a_c is the number of distinct applications using the class c. For example, the `Liquor` classes have the value of 2 because they are used from `WarehouseApp` and `StoreApp`.

The universality of a class c is defined as follows:

$$\text{universality}(c) = \frac{\log(i_c + 1)}{\log(i_{\max})} \times \frac{\log(a_c + 1)}{\log(a_{\max})}$$

where i_c is the class fan-in of c; $i_{\max}$ is the max value of the class fan-in of all classes; a_c is the application fan-in of c; $a_{\max}$ is the max value of the application fan-in of all classes. This metric is designed so that classes used in various applications have a high value. We employ log value instead of raw value because the distribution of the fan-in follows the power-law, i.e., a small number of classes have extremely large fan-in meanwhile almost all classes have small fan-in [3].

3.3 Pattern universality

We define *pattern universality* based on the universality metric. The pattern universality of a pattern p is the minimum value of the universality values of the classes whose methods are invoked in the pattern p. If the pattern universality of a pattern goes over the threshold value k, the pattern is filtered out as a generic pattern. Since method calls contained in a Fung's coding pattern do not have class name, classes used in a pattern are acquired from its instances in the original source code.

4 Case studies

We have set up two case studies to evaluate our analysis method. In Case Study 1, we have collected various open source software packages and measured the universality for all classes. Based on the result of the Case Study 1, Case Study 2 measured pattern universality for each coding pattern detected by Fung to filter out the coding patterns.

4.1 Case Study 1

In order to measure and observe the universality value for actual classes, we collected the source files of 39 application packages involving 131,328 classes; the total LOC is 18,778,821. The applications include Java SE API and various open source software packages covering a broad range of domains in addition to applications analyzed in [5]. For example, the applications include Eclipse 3.3 (IDE), Netbeans 5.5.1 (IDE), jEdit 4.3 (Text editor), Azureus 3.0.3.4 (Network Client), Apache Tomcat 6.0.14 (Network Server), Spring framework 2.5.5 (Application framework), Freemind 0.8.1 (Drawing), JHotDraw 7.0.9 (Drawing), HSQLDB 1.8.0 (Database) and so on.

We analyzed the cross-application use-relation of the applications and measured the universality for all classes. We have investigated following data in the case study.

Table 1. Top 20 classes in the universality

Rank	Class name	Univ.	Fan-in
1	java.lang.String	0.933	69,324
2	java.lang.Object	0.915	55,628
3	java.util.List	0.793	12,981
4	java.lang.System	0.780	11,191
5	java.lang.Class	0.776	10,590
6	java.lang.Throwable	0.775	10,467
7	java.util.Iterator	0.773	10,191
8	java.util.ArrayList	0.772	10,135
9	java.lang.Exception	0.761	8,840
10	java.util.Map	0.757	8,476
11	java.lang.Integer	0.748	7,568
12	java.util.Set	0.741	6,954
13	java.io.File	0.736	6,554
14	java.lang.StringBuffer	0.735	6,907
15	java.io.PrintStream	0.730	6,132
16	java.util.HashMap	0.730	6,129
17	java.io.IOException	0.725	6,115
18	java.util.Collection	0.724	5,690
19	java.lang .IllegalArgumentException	0.714	5,057
20	java.lang.Runnable	0.699	6,790

The top-20 classes in the universality We can see what types of classes have high universality value by universality ranking.

Difference between the universality and the fan-in In order to observe whether or not the universality can distinguish the general-purpose classes precisely than the raw fan-in, we present classes whose universality is high but fan-in is low and classes whose universality is low but fan-in is high, respectively.

Distribution of the universality This list indicates threshold that separates general-purpose classes, domain-specific classes and application-specific classes.

Result The top-20 classes in the universality are listed at Table 1. We can see that the fundamental classes of the Java SE API have especially high values, followed by the general-purpose utilities such as collection-related classes. We guess that almost all Java programmers have used the listed classes.

Table 2 lists five of the 47 classes that the order (ranking) in the universality < 100 meanwhile the order in the fan-in > 100. We can see that all of the classes (including the omitted ones) belongs to Java SE and have fundamental/utility role. For example, `LinkedList` and `Stack` provide data model and operations for collection manipulation.

On the other hand, the classes that have high fan-in but low universality includes the classes for application-

Table 2. Classes with high universality but low fan-in

Class name	Univ. Rank	Fan-in Rank
java.lang.Character	39	104
java.util.LinkedList	41	105
java.io.FileOutputStream	56	177
java.lang.Comparable	78	240
java.util.Stack	95	354

Table 3. Classes with high fan-in but low universality

Class name	Univ. Rank	Fan-in Rank
org.eclipse.swt.widgets.Control	213	25
org.eclipse.swt.SWT	221	34
org.eclipse.core .resources.IResource	564	69
org.openide.util.NbBundle	1,398	24
org.openide.ErrorManager	1,496	54

specific crosscutting concerns such as resource management (`NbBundle`, `IResource`) and error management (`ErrorManager`), as listed in Table 3. We found 47 classes that are out of the top 100 of universality ranking in the top 100 of fan-in ranking. These classes have high class fan-in value because they are frequently used in a large application (e.g., Eclipse or NetBeans), but have low universality because they are used from a few applications.

This result shows that it is required to consider the application border for measuring how widely a class is used because classes implementing crosscutting concerns in a large application occasionally have large fan-in.

Table 4 presents the distribution of the universality value. The general-purpose classes exist at the range of 1.0–0.5. The domain-specific classes such as GUI, networking utility or logger are found at the range of 0.5–0.2. The classes used locally appears at the range of 0.2–0. Therefore, the threshold $k = 0.5$ seems appropriate to filter the generic patterns, $k = 0.2$ seems appropriate to filter the domain-specific patterns, respectively.

4.2 Case Study 2

We apply our method to coding patterns detected by Fung [5] in order to evaluate the filtering ability of the universality metric. We analyzed the patterns of Azureus, Apache Tomcat and SableCC. The pattern universality is measured using the same application collection used in the Case Study 1. We categorized patterns into three categories according to their pattern universality values. We refer to patterns whose universality are higher than 0.5 as *generic* patterns, patterns whose universality are in the range of 0.5 and 0.2 as *domain-specific* patterns and classes whose universality are lower than 0.2 as *application-specific* patterns respectively.

Result — Azureus We found the text manipulation patterns such as combination of the `String.substring()` and `String.indexOf()` and the collection manipulation patterns such as the iterator idiom as generic patterns.

As domain-specific patterns, we found I/O classes in the `java.nio` package and the collection manipulation using `LinkedHashMap`, that is an implementation of Map interface with an additional feature.

The application-specific patterns of Azureus include logging patterns using application-local `Debug` class and a synchronization process with a pair of `AEMonitor.enter()` and `AEMonitor.exit()`.

Eight of ten Azureus's patterns reported in [5] are categorized to application-specific patterns; the other patterns are recognized as a generic pattern because both of them are error management concerns comprising `List`, `Iterator` and `Exception`, who are found in Table 1.

Result — Apache Tomcat The generic patterns of Tomcat are similar to the ones of Azureus. There are text manipulation and collection manipulation.

We found JDBC-related patterns using classes of `java.sql` package and I/O patterns using `BufferedOutputStream` as domain-specific patterns.

The application-specific patterns of Tomcat include logging concerns using application-local `Log` classes and HTTP processing with `Http11NioProtocol`.

Ten of eleven patterns of Tomcat reported in [5] are distinguished as application-specific; the other pattern is categorized to domain-specific pattern because it consists of `ResourceBundle`, that is a resource management class in Java SE.

Result — SableCC The patterns filtered as generic patterns include `Map` and `List` operations.

As a domain-specific pattern, we found a pattern using `ListIterator`, a variation of iterator which has some additional operation such as `previous()`.

We found syntax tree manipulation patterns using `Tree` or `Token` as application-specific patterns.

The all seven patterns reported in [5] are distinguished as application-specific patterns.

4.3 Discussion

Using the universality metric, we succeeded to distinguish the generic concerns such as string/collection manipulation concerns and the domain-specific patterns such as

Table 4. Distribution of universality

Univ.	#classes	packages
1.0–0.9	2	java.lang
0.9–0.8	0	(none)
0.8–0.7	17	java.util, java.lang, java.io
0.7–0.6	18	java.lang, java.util, java.io, java.net, java.awt
0.6–0.5	49	java.util, java.lang, java.io, javax.swing, java.awt,...
0.5–0.4	80	java.io, java.lang, javax.swing, javax.swing, java.awt,...
0.4–0.3	196	org.eclipse.swt.widgets, javax.swing, java.util, java.awt.event, java.lang, ...
0.3–0.2	348	org.eclipse.swt.widgets, org.eclipse.swt.graphics, javax.swing, javax.management, java.awt, ...
0.2–0.1	1,385	org.eclipse.swt.widgets, org.eclipse.swt.dnd, javax.management, org.gudy.azureus2.core3.util, org.bouncycastle.asn1, ...
0.1–0	96,604	soot.jimple.parser.node, org.apache.poi.....functions, test, soot.coffi, ...

database connection patterns. It is notable that the universality metric prevents that the classes used in a few huge applications such as Eclipse are accidentally categorized into generic classes. Our method can provide configurable filtering of concerns without deep domain- or application-knowledge: high threshold value filters out generic concerns; low threshold value enables to recognize application-specific concerns.

An interesting point of our universality is that some domain-specific class may have higher class universality than general-purpose classes. For example, `java.awt.Component` has high universality value of 0.632 although it is a GUI-related class; on the other hand, `java.util.LinkedHashMap` and `java.util.ListIterator` have low universality value of 0.358 and 0.467 respectively although they are general-purpose collection-related classes. Although this may not be a matter because famous domain-specific concerns are filtered out and less-popular generic concerns are not filtered out, we have several ideas to improve distinguishing ability of our method. For example, treating a method call against an overridden method as method calls to all overriding/overridden methods as Marin et al. proposed [6] may be effective to find a universally-used class hierarchy such as collection classes. Another idea is using our method with other method or metrics such as utilityhood [2], that measures how a method is likely to be a utility.

Universality metric value depends on a set of applications. Although we have collected famous open source software, we need further case study in different target, e.g. applications developed in an industry.

5 Conclusions

In this paper, we propose cross-application fan-in analysis to automatically filter generic or application-independent concerns from coding patterns, or crosscutting concern candidates. Our method is constructed for Fung's pattern mining; however, we believe that our method works with other crosscutting concern detection methods. Combining with such methods is a future work. We are also planning to improve our analysis method as discussed in4.3

Acknowledgements This work has been supported by Japan Society for the Promotion of Science, Grant-in-Aid for Exploratory Research (18650006), and the Microsoft IJARC CORE4 Project, and has been conducted as a part of Stage Project, the Development of Next Generation IT Infrastructure.

References

[1] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Software Eng.*, 31(10):804–818, Oct. 2005.

[2] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proc. 14th Intl' Conf. Program Comprehension (ICPC 2006)*, pages 181–190, June 2006.

[3] M. Ichii, M. Matsushita, and K. Inoue. An exploration of power-law in use-relation of java software systems. In *Proc. 19th Australian Software Eng. Conf. (ASWEC 2008)*, pages 422–4311, Mar. 2008.

[4] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking significance of software components based on use relations. *IEEE Trans. Software Eng.*, 31(3):213–225, Mar. 2005.

[5] T. Ishio, H. Date, T. Miyake, and K. Inoue. Mining coding pattern to detect crosscutting concerns in java programs. In *Proc. 15th Working Conf. Reverse Eng. (WCRE 2008)*, pages 123–132, Oct. 2008.

[6] M. Marin, A. V. Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Trans. Software Eng. and Methodology*, 17(1):3, Dec. 2007.