Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2017-IC-007

2017-IC-007

Are Your Classes Well-Encapsulated? Encapsulation Analysis for Java

Zhenhao Tang, Juan Zhai, Bin Li, Jianhua Zhao

International Conference on Software Quality, Reliability and Security 2017

2017 IEEE International Conference on Software Quality, Reliability and Security

Are Your Classes Well-encapsulated? Encapsulation Analysis for Java

Zhenhao Tang, Juan Zhai, Bin Li, Jianhua Zhao
State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
Nanjing, Jiangsu, China 210023
tangzh@seg.nju.edu.cn, zhaijuan@nju.edu.cn, hsslb@seg.nju.edu.cn, zhaojh@nju.edu.cn

***Abstract*—Encapsulation is one of the basic characteristics of object-oriented programming. However, the access modifiers provided by common object-oriented languages do not help much because they only encapsulate the member references rather than the objects pointed to by them. Bad encapsulation makes object-oriented programs difficult to understand and reason about, thus concealing potential software vulnerabilities. We present in this paper the encapsulation analysis technique, which is an expression-based dataflow analysis, to statically compute the runtime memory layouts of object-oriented programs. The analysis results can help developers to master an intuitive comprehension on the code quality regarding encapsulation of classes. The results of experiments on various open-source Java projects and libraries show that our approach is both effective (25.76% of the classes are reported as not fully encapsulated) and efficient (2.15 KLOC/s and 27.35 classes/s) in finding potential encapsulation problems. We also give common guidance on how to achieve better encapsulation for object-oriented programs.**

I. Introduction

Encapsulation is one of the basic characteristics of object-oriented programming (OOP). It is a language facility designed to hide information, reduce system complexity and increase robustness. Basically, it controls the access to an object's members and enables the bundling of data with methods operating on it. Mainstream OOP languages like C++ and Java provide several access modifiers (e.g. `private`, `protected` and `public`) to limit the way in which the outside can access a class's members. However, access modifiers cannot indeed protect inner members because they only encapsulate the references (or pointers) rather than the objects they point to. Consequently, class members are often exposed by either capturing or leaking. Capturing means the inside of a class depends on the outside. For example, setter methods and constructor methods that assign their arguments to private members, generate aliases from inside to outside. Leaking means the inside of a class is exposed to the outside. For instance, naive getter methods which directly return a member of a class easily open the accessibility of non-public fields to the outside.

In fact, the real encapsulation problem depends on the topology of objects and object references in the memory. It is an alias problem, which usually results in unintentional security holes, mistaken assumptions or side effects. A class is fully encapsulated if all its members and the objects they point to cannot be modified by the outside without calling its methods. Hence, the state of an object can be changed only by the methods provided by the class. In this sense, the correctness of OO programs often relies on good designs and implementations of encapsulation. For example, some class methods are supposed to return copies of their inner data rather than just copies of references. Therefore any modification on the returned objects will not affect the internal states of the objects. Determining how well-encapsulated the classes are is critical in both understanding and reasoning about OO programs.

To help developers to master a clear and intuitive understanding of their programs, we introduce in this paper an encapsulation analysis technique to discover encapsulation properties. It is an expression-based dataflow analysis that computes the runtime memory layout abstractions for OO programs. Its analysis results identify the classes that have potential encapsulation problem, the class members that are shared and the methods that expose the members. Besides, our approach indicates how the shared memory is accessed by the outside.

Our approach is designed and implemented for Java, which is a relatively pure OO style language and there is only reference semantics for objects. It can be ported to other object-oriented languages by considering their own semantics w.r.t. encapsulation.

Our main contributions are as follows.

- We propose a static analysis technique that generates results which help developers find classes with potential poor encapsulation. It is based on expressions and is therefore more flexible and precise than those based on statements. It is also more intuitive for developers to locate problems than those based on static single assignments (SSAs) or other intermediate representations.
- We implement a prototype which finds real encapsulation problems for various open-source Java projects and libraries. We find that about 25.76% of the total 5963 classes potentially share memory with the outside. The average throughput our prototype is 2.15 KLOC/s and 27.35 classes/s.
- We give guidance on how to review and refactor code to increase encapsulation based on the results of encapsulation analysis, with less human efforts while ensuring functional equality.

978-1-5386-0592-9/17 $31.00 © 2017 IEEE
DOI 10.1109/QRS.2017.31

II. Motivation

The motivation of our work is to use static analysis to discover potential encapsulation problems.

1) Basic idea: Poor encapsulation brings in potential bugs. The rationale contains two parts: first, the target class is poorly encapsulated and shares its inner memory with the outside through its methods; second, the outside modifies the shared memory, resulting in unintentional behaviors that are unknown and invisible to the class. Our approach deals with this problem from two perspectives:

- From the viewpoint of the classes, we identify the classes that potentially share memory with the outside, the members that are shared, and the methods are responsible.
- From the viewpoint of the clients, we discover how the shared memory is accessed by the outside.

2) Example: The program shown in Fig. 1 is a simple implementation of the structure sharing iterator design presented by Noble et al. [20]. From the viewpoint of the class `LinkedList`, its member `first` is exposed to the class `SharingIterator` through the method `sharingIterator()`, in which the `new` expression in line 44 calls the constructor of `SharingIterator`. Hence, the object pointed to by an instance of `LinkedList` is captured by an instance of `SharingIterator`. As a result, the class `LinkedList` has an ecapsulation problem which makes its member `first` leak. Any modification that happens to the leaked object potentially makes the states of its instances invalid. But the class `LinkedList` has no idea what will happen outside.

On the other hand, from the viewpoint of the class `SharingIterator`, it captures a member of `LinkedList` through its constructor. Then `current` points to the same object as `first` in `LinkedList` does. The real problem occurs in line 21, where `null` is assigned to the field `next` of `current`, which breaks the inner structure of `LinkedList` Then an instance of `LinkedList` becomes invalid and further operations on it will result in unpredictable behaviors. Note that the method `bad` is an error that is deliberately inserted and the original methods in `SharingIterator` is fine because they neither modify the data structure nor further leak `current` to other classes. This example illustrates that a less strict encapsulation discipline in design or implementation can bring in potential vulnerabilities and real bugs can happen.

III. Encapsulation Analysis

In this section, we introduce encapsulation analysis, which computes statically the runtime memory layouts representing encapsulation and ownership topology. Encapsulation analysis is an expression-based dataflow analysis. It consists of both intra- and inter-procedural parts. It uses some concepts of escape analysis, for example, escapement. But the concept is extended to a broader meaning. Escape analysis only cares whether an object escapes from the method or thread in which it is created. Encapsulation analysis works on a higher level: classes and objects. It concerns whether class members escape from the classes where they belong.

```
1  public class SharingIterator implements Iterator {
2      private Node current;
3      SharingIterator(Node head) {
4          current = head;  // capturing
5      }
6      @Override
7      public boolean hasNext() {
8          return current != null;
9      }
10     @Override
11     public Object next() {
12         if (hasNext()) {
13             Object ret = current.value;
14             current = current.next;
15             return ret;
16         }
17         return null;
18     }
19     public void bad() {
20         current.next = null;  // Oops!
21     }
22 }
23
24 public class LinkedList implements List {
25     private Node first = null;
26     private Node last  = null;
27     private int  size  = 0;
28     @Override
29     public boolean add(Object o) { /* ... */ }
30     @Override
31     public void clear() { /* ... */ }
32     @Override
33     public Object get(int index) { /* ... */ }
34     @Override
35     public Object set(int index, Object element) {
36         /* ... */
37     }
38     @Override
39     public int size() { /* ... */ }
40     public Iterator sharingIterator() {
41         return new SharingIterator(first);  // leaking
42     }
43 }
```

Fig. 1: An implementation of the structure sharing iterator

A. Memory Graph Abstraction

We first introduce a static abstraction that simulate the runtime memory topology. We call it memory graph, which is designed to represent typical OOP features. Primitive types (e.g. `int`, `boolean` etc.) are ignored in our abstraction, because they have only value semantics in Java and they don't has aliases.

Definition 1. A memory graph is a directed graph $MG = (N_c \cup N_t \cup N_p \cup N_r \cup N_f \cup N_l, E_f \cup E_s \cup E_o \cup E_p \cup E_r)$, where

- N_c represents the set of objects created by class instance creation expressions. We call them *created objects*.
- N_t represents the set containing the current class instance, outer class instances, and super class instances. We call them *this objects*.
- N_p represents the set of objects that are peers (objects of the same type) of the current instance. We call them *peer objects* for simplicity.
- N_r represents the set containing the references `this`, `outer` and `super`, pointing to the current class instance, outer class instances and super class instances respectively.

- N_f represents the set of field references.
- N_l represents the set of local references.
- E_f represents the set of field edges. If $x \xrightarrow{f} y \in E_f$, then $x \in N_c \cup N_t \cup N_p$ and $y \in N_f$.
- E_s represents the set of super class field edges. If $x \xrightarrow{s} y \in E_f$, then $x \in N_c \cup N_t \cup N_p$ and $y =$ `super`.
- E_o represents the set of outer class field edges. If $x \xrightarrow{o} y \in E_f$, then $x \in N_c \cup N_t \cup N_p$ and $y =$ `outer`.
- E_p represents the set of points-to edges. If $x \xrightarrow{p} y \in E_p$, then $x \in N_r \cup N_f \cup N_l$ and $y \in N_c \cup N_t \cup N_p$.
- E_r represents the set of refers-to edges. If $x \xrightarrow{r} y \in E_r$, then $x \in N_r \cup N_f \cup N_l$ and $y \in N_r \cup N_f \cup N_l$.

In our memory graph abstraction, a node in usually corresponds to one or more AST nodes. A node in N_c corresponds to a class instance creation expressions (`new` expressions), a node in N_t corresponds to a type (class) declaration and a node in N_p corresponds to a parameter definition of the same type as the current instance. A node in N_l corresponds to one or more expressions that resolve to references. A node in N_f corresponds to a name that is resolved to be a field. The only exceptions are the references in N_r. In Java, the use of keyword `this` is not necessary in many cases when members of the current instance are accessed when there is no ambiguity. Similarly, the keyword `super` can be omitted when members in super class instances are accessed without ambiguity. In Java, there is no such keyword as `outer`. So nodes in N_r do not necessarily correspond to AST nodes. All these kinds of references are implicitly used according to the context.

Note that N_c, N_t and N_p are sets of objects. In later graphic illustration, objects are indicated by boxes in which node types are labeled as `created`, `this` and `peer` respectively. N_r, N_f and N_l are sets of references which are represented by circles. References in N_r are indicated by dash circles and references in N_f and N_l are indicated by solid circles. Refers-to edges are shown in dashed line while other edges are shown in solid lines.

Also note that, there is only one current instance and one `this` reference in a memory graph. This is straightforward based on the concept of OOP. But there can be multiple outer class instances, super class instances and references pointing to them, because of the hierarchical structures of inner-outer classes and inheritance. For example, `B` is the outer class of `A` and `C` is the outer class of `B`. From the perspective of `A`, in order to access a member of class `C`, we need two outer class instances (`B` and `C` respectively) and two `outer` references (one from `A` to `B` and another from `B` to `C`).

Every non-static method of a class has an extra implicit parameter representing the actual instance bound to it. This method can access any member of the object. To model this feature, we build a memory graph for each class declaration. Then, upon entry to a method, we add the memory graph of the belonging class to its initial memory graph.

Example 1. Fig. 2(a) shows a Java program containing declarations of five classes: `Bat`, `Baz`, `Bar`, `Foo` and `Dom`. `Bar` is the base class of `Foo` and `Dom` is the inner class of `Foo`. `Dom` has two members of the type `Bat` and `Baz` respectively. Fig. 2(b) and 2(c) show the memory graphs of the method `Dom(Dom)` (the copy constructor of the class `Dom`) before and after the code block executes, respectively.

Because `Dom(Dom)` is a non-static method of class `Dom`, it contains the memory graph of the class `Dom`, which is the lower connected graph in Fig. 2(b). The memory graph of `Dom` has an implicit `outer` field which points to the memory graph of `Foo`, because it is an inner class of `Foo`. Similarly, the memory graph of `Foo` contains an implicit field `super` pointing to the memory graph of `Bar`. At the top of Fig. 2(b) is a connected graph built once the analysis enters the method `Dom(Dom)`. There is a peer object of `Dom` pointed to by the reference `other` representing the parameters of the method. The detailed process of building a memory graph for a class is shown in III-C3.

We can see that there are two more refers-to edges in the upper connected graph of Fig. 2(c) then that of Fig. 2(b). This is due to the effect of the method body {`this.bat = other.bat; this.baz = other.baz;`}. The detailed process of building a memory graph for a method is described in III-C4. □

B. Escapement

We define four kinds of escapement for nodes (including objects and references): `ArgEscape`, `RetEscape`, `GloEscape`, and `NoEscape`. Note that we do not strictly distinguish between objects and references w.r.t. escapement for simplicity. The definitions of escapement are as follows.

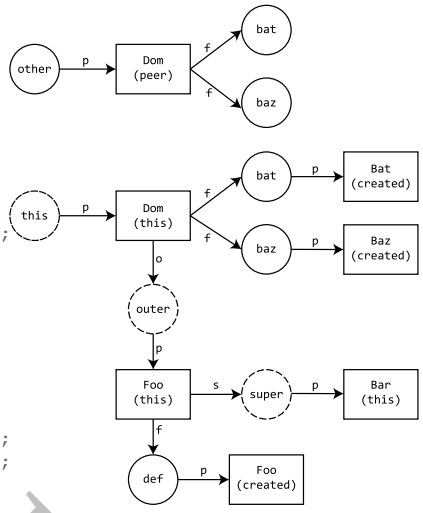
Definition 2. Let O be an object or a reference, M be a method and C be a class. O is said to escape M by arguments, denoted as $ArgEscape(O, M)$, if O can be reached from any argument in M. O is said to escape C by arguments, denoted as $ArgEscape(O, C)$, if O escapes any method of C by arguments.

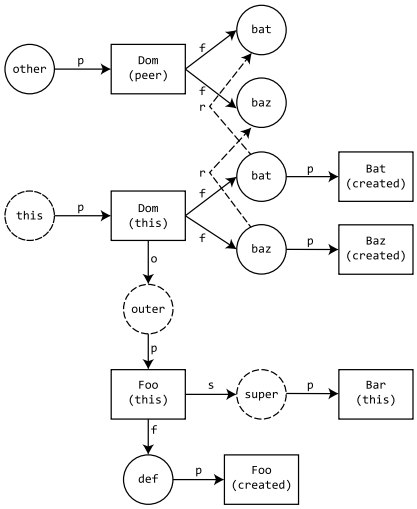
Definition 3. Let O be an object or a reference, M be a method and C be a class. O is said to escape M by return, denoted as $RetEscape(O, M)$, if O can be reached from any return value of M. O is said to escape C by return, denoted as $RetEscape(O, C)$, if O escapes any method of C by return.

Definition 4. Let O be an object or a reference and C be a class. O is said to escape C globally, denoted as $GloEscape(O, C)$, if O can be reached by static or public fields of any class.

Definition 5. Let O be an object or a reference, M be a method and C be a class. O is said to not escape M, denoted as $NoEscape(O, M)$, if $\neg ArgEscape(O, M)$ and also $\neg RetEscape(O, M)$. O is said to not escape C, denoted as $NoEscape(O, C)$, if $\neg ArgEscape(O, C)$, $\neg RetEscape(O, C)$ and $\neg GloEscape(O, C)$.

```
1  class Bat { /* ... */ }
2  class Baz { /* ... */ }
3  class Bar {
4      String str = null;
5      public Bar(String str) {
6          this.str = str;
7      }
8  }
9  class Foo extends Bar {
10     static Foo def = new Foo("");
11     public Foo(String str) {
12         super(str);
13     }
14     class Dom {
15         Bat bat = new Bat();
16         Baz baz = new Baz();
17         Dom() {}
18         Dom(Dom other) {
19             this.bat = other.bat;
20             this.baz = other.baz;
21         }
22     }
23 }
```





(a) A Java program (b) Memory graph at line 18 of `Dom(Dom)` (c) Memory graph at line 21 of `Dom(Dom)`

Fig. 2: An example of the memory graph abstraction

C. Intraprocedural Analysis

Our dataflow analysis is flow-insensitive and forward directed, based on a standard iterative scheme. An notable characteristic of our analysis is that it is based on expressions rather statements. To be precise, our flow functions deal with each expression in the abstract syntax trees (ASTs). This is more fine-grained that those based on statements because it is flexible enough to deal with various kinds of combinations of expressions while statement-based approaches need to enumerate all kinds of possible statements, which is tedious and imprecise. This also means we use expression-based control flow graphs (CFGs) to conduct our dataflow analysis.

1) Dataflow Equations: Given a node n in the expression-based CFG of a method, a memory graph at the entry to n (denoted as G_i^n) and the memory graph at the exit from n (denoted as G_o^n) are related by the standard dataflow equations:

$$\begin{aligned} G_o^n &= f(G_i^n) \\ G_i^n &= \wedge_{m \in pred(n)} G_o^m, \end{aligned}$$

where f is the flow function. Note again that m is a predecessor of n, and they are connected through expressions rather than statements.

The meet (or merge) operation between two memory graphs $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$ is defined as the union of the two graphs: $G_1 \wedge G_2 = (N_1 \cup N_2, E_1 \cup E_2)$.

The merge of two memory graphs consists of two parts. First, we need to merge the nodes. This is a set union operation on the sets of nodes. If two nodes represent the same AST nodes in two graphs, we need to compute the new escapement values of the merged nodes according to the lattice of escapement described before. Second, we need to merge the edges. This is a standard set union operation on the sets of edges.

2) Correlations between Nodes and Expressions: The correspondance between graph nodes and expressions (AST nodes) is necessay in updating the memory graphs according to program expressions. That is, we need to find the graph nodes that correspond to target expressions.

The way to find the associated reference node of an expression `exp` is as follows:

- If `exp` is a class instance creation expression of the form `new T(exp1)`, we search the memory graph for the node that corresponds to this expression. The correlated object node will be created if it does not exist in the graph.
- If `exp` is a simple name, we search the memory graph by matching this name. The correlated reference node will be created if it does not exist. A static or public field reference is marked as `GloEscape`. A parameter reference is marked as `ArgEscape`. Other references are marked as `NoEscape` by default.
- If `exp` is a qualified name or a field access like `p.f`. If `p` is a simple name, we first find the reference node r that corresponds to `p`. Then we get the field node named `f` of r and put it in the set F. Then F is the set of reference nodes that correspond to `p.f`. If `p` is a qualified name, we first find the set O of nodes that correspond to `p`. Then for each node o in O, we get the field node of o, and put it in F. Then F is the set of reference nodes that correspond to `p.f`.
- If `exp` is an assignment of the form `exp1=exp2`. The set of reference nodes that correspond to the assignment is actually the one of `exp1`. In this case, `exp1` must be an lvalue and `exp2` must be an rvalue.
- If `exp` is a method invocation expression of the form `exp1.foo(exp2)`. When the return type of method

foo is void, the expression does not have a correlated node in the memory graph. If the return type is not void, the corresponding set of reference nodes is the one that corresponds to the expression obtained by instantiating the returned expression of the callee by replacing parameters with arguments. The details of instantiation will be introduced in III-D2.

- If exp is a cast expression (like (Foo)exp1) or a parenthesized expression (like (exp2)), the sets of corresponding nodes are the sets of nodes that correspond to exp1 and exp2 respectively.
- If exp is a simple this expression of the form this, it corresponds to the unique reference node this in the memory graph. If the expression is of the form Foo.this, it corresponds to an outer reference or a super reference in the memory graph, depending on whether Foo is the outer class or the super class of the current class. It can also be a nested expression like Foo.Bar.this where Bar is a outer class and Foo is a outer class of Bar. The accurate corresponding set of nodes is determined by class relationship diagrams including inner-outer relationship and inheritance relationship.

3) Building Memory Graphs for Classes: The intial memory graph $MG = (\emptyset, \emptyset)$. We first add a reference called this and an object labeled C(this) (where C is the class name) to the node set. Then a points-to edge from this to C(this) is added to the edge set.

If the class has an outer class, we then create an outer reference and an object called O(this) (where O is the class name of the outer class). Also, an outer class field edge is added from C(this) to outer, and a points-to edge is added from outer to O(this). The process of creating super class references, super class objects and super class field edges is similar to that for outer classes, except that the reference node is called super instead of outer. This process can repeat several times if the outer class has its own outer class or the super class has its own super class.

Two more program constructs are dealt with here: initializers that come with field declaration (e.g. T f = new T();) and instance initialization blocks (e.g. {g = new S();}). Both these two cases are deal with in the same way as we deal with method bodies, which will be introduced later in III-C4.

4) Building Memory Graphs for Methods: Upon entry to a method, our analysis first copies the memory graph of its belonging class as the initial memory graph of the method. Then, this memory graph is updated by two kinds of program artifacts: 1) parameter definitions and 2) method bodies.

For each parameter definition, if the parameter type is non-primitive, we add to the memory graph a reference node r, an object node o, and a points-to edge from r to o.

For method bodies, we identify the following basic expressions in the bodies that could affect our analysis results, together with their flow functions. Note that the effects of sophisticated combination of expressions are computed recursively.

- exp1 = exp2: We first search the current memory graph for the set $R1$ of corresponding reference nodes of the lvalue exp1 and the set $R2$ of corresponding reference nodes of the rvalue exp2. Then, for each reference node $r1 \in R1$ and $r2 \in R2$, we add a refers-to edge from $r1$ to $r2$.
- T exp1 = exp2: The effect of a declaration with an initializer is just like the effect of the assignment exp1 = exp2.
- new T(exp): We first create a created object node (in N_c) corresponding to new T(exp) if it does not exist in the current graph.
- exp1.foo(exp2): We instantiate the summary graph (which will be explained in III-D1) of the method foo and attach it to the current graph. The escape states of the nodes correponding to exp2 are updated to ArgEscape. The algorithm of instantiation will be introduced later in III-D2.
- return exp: We find the set of reference nodes corresponding to exp and update the escape state of each node in it with RetEscape (according to the escapement lattice).

D. Interprocedural Analysis

We now introduce the interprocedural part of our analysis. Our analysis is summary-based and modular. Method are analyzed and the analysis results are saved. When encountering a method invocation, we instantiate the summary information at the call site to obtain the effects of the invocations, without going into the body of the invoked methods.

1) Summary Graphs: At the core of our interprocedural analysis are summary graphs, which represent the summary information of analyzed methods. The summary graph of a method is a subgraph of its memory graph. The computation of the summary graph of a method consists of two steps: 1) update the escape states of all nodes in the memory graph using Algorithm 1; 2) find all the nodes whose escape states are either ArgEscape or NoEscape, or the nodes that are reachable from this node which is part of the memory graph of its belonging class. All these nodes together with the related edges compose the summary graph.

2) Instantiation of Summary Graphs: We use the algorithm presented by Choi et al. [6] to update the caller's memory graph according to the callee's summary graphs. We only introduce the basic ideas of the algorithm here. Suppose that a_i is a node in the callee's memory graph, $\hat{a_i}$ is the corresponding node of a_i in the caller's memory graph. Then, updating the caller's memory graph consists of two steps: 1) updating the node set of the caller's memory graph using a_i and $\hat{a_i}$; 2) updating the edge set of the caller's memory graph using a_i and $\hat{a_i}$.

To update the caller nodes with the callee nodes, we need to find the corresponding relation in two memory graphs recursively. As a base case, a_i maps to $\hat{a_i}$. For the inductive

Algorithm 1: UpdateEscapeStates

```
1  Function UpdateEscapeStates(mg,es)
       Input : A memory graph mg, an escape state es
       Output: mg with updated escape state es
2      begin
3          worklist ← ∅
4          nodes ← the set of nodes in mg
5          foreach n ∈ nodes do
6              if n.escape==es then
7                  add n to worklist
8          while worklist ≠ ∅ do
9              remove a node m from worklist
10             ← outgoing nodes of m
11             foreach o ∈ do
12                 if o.escape<es then
13                     o.escape=es
14                     add o to worklist
15 Function UpdateEscapeStates(mg)
       Input : A memory graph mg
       Output: mg with updated escape states
16     begin
17         UpdateEscapeStates (mg, GloEscape)
18         UpdateEscapeStates (mg, ArgEscape)
19         UpdateEscapeStates (mg, RetEscape)
```

step, the nodes pointed to by a_i should map to the nodes pointed to by $\hat{a_i}$. Then, the escape state of the caller node is updated ty the callee node. If there is no corresponding node in the caller's memory graph, we create on with the same escape state of the node in the callee's memory graph.

To update the caller edges with the callee edges, for each edge in the callee's memory graph, we first find its two connected nodes. Finally, we find the two corresponding nodes in the caller's graph. Then we add the edge between the two nodes if it does not exist.

IV. Evaluation

To evaluate the usefulness of our approach, we answer the following research questions:

- RQ1: How effective is encapsulation analysis?
- RQ2: Is encapsulation analysis efficient enough to process real-life projects?
- RQ3: Do the analysis results help developers understand and review code?

RQ1 cares how many real encapsulation problems can be found by our approach. RQ2 concerns the efficiency of analysis, especailly on large scale of code bases. RQ3 concerns whether the analysis results and the way in which they are presented help developers improve code quality regarding encapsulation.

A. Setup

We implement our approach as part of a Hoare-style code verification tool called Accumulator written in Java. The tool and the source code of encapsulation analysis can be found here[1]. It uses Eclipse Java development tools (JDT) to generate ASTs, based on which expression-based CFGs are built. Then our dataflow analysis is performed on these CFGs.

All the experiments were conducted on a PC with an Intel i7-4790 (3.6 GHz) CPU and 16 GB DDR3 RAM, running the Windows 10 operating system and Java SE Runtime Environment 1.8.0_65.

We chose various open-source projects and libraries (ranging from 6.7k to 124.2k lines of code) to evaluate our approach, shown in the first column of Table I. Project Avrora is a set of simulation and analysis tools for programs written for the AVR microcontroller produced by Atmel and the Mica2 sensor nodes by the UCLA Compilers Group. The Guava project contains several of Google's core libraries that we rely on in our Java-based projects: collections, caching, primitives support, concurrency libraries, common annotations, string processing, I/O, and so forth. ICTCLAS4j is a Chinese lexical analysis system written in Java by Institute of Computing Technology, Chinese Academy of Sciences. We also choose the most commonly used two packages of the Java Development Kit (java.lang and java.util) as our experiment target. JPaul is a Java program analysis utilities libraries based on Ant and JUnit. MySQL Connector/J is the Java database connectivity tool for the MySQL database. Sunflow is an open source rendering system for photo-realistic image synthesis. It is written in Java and built around a flexible ray tracing core and an extensible object-oriented design. Finally, we also choose the ownership collections (OC) [23], which contain the original JDK 5.0 collection classes and several variants rewritten to achieve a stronger ownership discipline.

Project	Overview				
	SLOC	CUs	Classes	Methods	Time (s)
Avrora	70.1K	487	1753	8941	46.7
Guava	74.8K	514	1120	10682	17.1
ICTCLAS4j	6.7K	34	34	412	2.0
JDK (lang, util)	124.2K	598	1211	13669	62.1
JPaul	41.3K	135	119	935	0.8
MySQL Connector/J	60.3K	264	269	6670	55.5
Sunflow	20.8K	169	183	1147	6.7
OC (Original)	56.7K	375	615	6558	21.2
OC (Memento)	10.4K	46	131	1600	2.0
OC (NoProxy)	10.6K	43	128	1723	1.9
OC (OasD)	10.5K	45	130	1638	2.0
Total	486.4K	2710	5963	53975	218.0

TABLE I: Projects and libraries used in our experiments

Columns 2 to 6 give the overview of these projects from several aspects, including source lines of code (SLOC), numbers of compilation units (CUs), numbers of classes, numbers of methods, and the time needed to be analyzed.

B. Results

[1]https://encap-analysis.github.io/encap-analysis/

1) Effectiveness: We list the experimental results for the projects and libraries in Table II. The results for each project or library consist of four categories: fully encapsulated classes, classes with `GloEscape` members, classes with `ArgEscape` members and classes with `RetEscape` members. Each category has two groups: the first group whose data is computed without considering the ownership semantics, and the second group whose data is computed with ownership semantics taken into consideration. Each group has two columns showing the numbers (No.) of classes and percentages (Pct.) w.r.t. the total numbers of classes in the projects respectively.

We can see that, without taking ownership semantics into consideration, 61.83% of the classes are regarded as encapsulated, which means no `GloEscape`, `ArgEscape` or `RetEscape` members are detected. 7.58% of the classes have at least one `GloEscape` member, 31.01% of the classes have at least one `ArgEscape` member and 12.06% of the classes have at least one `RetEscape` member. However, if we consider the ownership semantics, the numbers are very different. The percentage of total encapsulated classes increases to 74.24%, meaning that lots of false positives are filtered, according to our ownership semantics. Also, the numbers and percentages of classes with `ArgEscape` and `RetEscape` members decrease notably to 893 (14.98%) and 594 (9.96%) respectively. However, the numbers and percentages of classes with `GloEscape` members do not change, because none of the rules for ownership semantics has impact on public or static members.

To evaluate the rates of false positives and false negatives, we choose five relatively small projects to manually check the analysis results. The numbers and rates of false positives and false negatives are listed in Table III.

For each project, we list total numbers of classes, numbers of reported classes (Pos.) and numbers of non-reported classes (Neg.). We also list the manually checked numbers of false postives (F/P) and false negatives (F/N) together their rates. For these five projects, the average false positive rate is 14.60% and the average false negative rate is 1.92%.

2) Efficiency: For all the open-source projects and libraries listed, our approach processes a total number of 486.4k lines of code and 5963 classes in 218 seconds (not including the time for the generation of ASTs by JDT). On average, the throughput is 2.15 KLOC/s and 27.35 classes/s. The efficiency is achieved mainly because that our approach is modular and scalable for bigger projects. Note that the experiments are performed on a prototype with nearly no optimization.

V. Related Work

The idea of using types systems to enforce ownership comes from early work [2], [16]. After that, three main ownership type systems were proposed, including the Universe Type System [10], [11], [12], [13], [18], Ownership Types [7], [8], [9] and Ownership Domains [1]. Besides, Nägeli et al. [19] investigates that how three ownership type systems can be applied to design patterns. Noble et al. [21] tries to achieve balance between flexibility and encapsulation by allowing objects to be aliased but mitigating the undesirable effects. Östlund et al. [22] introduces a language that allows users to express ownership, uniqueness and immutability in their programs.

A typical related work is escape analysis [6], which finds out if an object escapes the method or thread in which it is created. Similar work includes points-to analysis [14] and alias analysis [15]. Another interesting work [3] proposes an approach for tracking static analysis violations over the revision history of a program and also for attributing the introduction and elimination of these violations to individual developers. However, there are too many false positives and developers usually don't like the way in which the analysis results are presented, just as the work by Johnson et al. [17] finds out.

Tufano et al. [24] studied when and why bad code smells are introduced into projects. The results show that code maintenance and evolution activities are probably the main reason, just as we expected. We also think that during the maintenance and evolution activities, the encapsulation disciplines for OO programs also decline, resulting in potential and unintentional memory problems. The focus of our work is to find such problems. The work by Balachandran et al. [4] is similar to ours in the sense that we both focus on reducing human efforts and improving code quality by static analysis. However, they emphasize on peer code review and how high-quality reviewer recommendation are generated. Zhang et al. [26] presents an interactive approach for inspecting systematic changes. It uses templates to generate similar changes in diff patches and detects potential mistakes. Barnett et al. [5] proposes a similar approach to help developers understand a code review by decomposing changesets.

VI. Conclusions and Future Work

We propose in this paper a static analysis technique called encapsulation analysis that statically computes the runtime memory layouts of OO programs, to determine how well-encapsulated the classes in the target programs are. The analysis results help developers to master an intuitive comprehension on the code quality of the classes regarding encapsulation. Good encapsulation help developers build confidence on their code, just like tests and verification, while bad encapsulation guides them to refactor their code to increase robustness and reduce potential vulnerabilities.

Experiment results show that our approach is effective (25.76% of the classes are found as not fully encapsulated) and efficient (2.15 KLOC/s and 27.35 classes/s on average) for eleven open-source projects and libraries. The evaluation also tells us that class members are more likely to be exposed by arguments than by return. Besides, we know that there are many `public` methods that potentially make the members shared with the outside, which is a big threat to code quality.

Our approach does not take libraries into consideration for now. Each time we encounter a method invocation whose method body (source code) is not available, we ignore its

Project	Encapsulated				GloEscape				ArgEscape				RetEscape			
	W/O Ownership		W/ Ownership		W/O Ownership		W/ Ownership		W/O Ownership		W/ Ownership		W/O Ownership		W/ Ownership	
	No.	Pct.	No.	Pct.	No.	Pct.	No.	Pct.	No.	Pct.	No.	Pct.	No.	Pct.	No.	Pct.
Avrora	1248	71.20%	1338	76.33%	420	23.96%	420	23.96%	469	26.75%	281	16.03%	92	5.25%	80	4.56%
Guava	741	66.16%	873	77.95%	0	0	0	0	321	28.66%	163	14.55%	174	15.54%	152	13.57%
ICTCLAS4j	20	58.82%	22	64.71%	2	5.88%	2	5.88%	12	35.29%	9	26.47%	10	29.41%	8	23.53%
JDK (lang, util)	710	58.63%	985	81.34%	1	0.01%	1	0.01%	484	39.97%	172	14.20%	167	13.79%	145	11.97%
JPaul	60	50.42%	71	59.66%	14	11.76%	14	11.76%	50	42.02%	37	31.09%	18	15.13%	16	13.45%
MySQL C./J	173	64.31%	193	71.75%	8	2.97%	8	2.97%	81	30.11%	56	20.82%	53	19.70%	51	18.96%
Sunflow	121	66.12%	126	68.85%	6	3.28%	6	3.28%	54	29.51%	49	26.78%	24	13.11%	23	12.57%
OC (Original)	401	65.20%	498	80.98%	1	0.16%	1	0.16%	195	31.70%	80	13.01%	100	16.26%	74	12.03%
OC (Memento)	74	56.49%	107	81.68%	0	0	0	0	50	38.17%	16	12.21%	27	20.61%	15	11.45%
OC (NoProxy)	69	53.91%	106	82.81%	0	0	0	0	54	42.19%	15	11.72%	27	21.09%	15	11.72%
OC (OasD)	70	53.85%	108	83.08%	0	0	0	0	56	43.08%	15	11.54%	27	20.77%	15	11.54%
Total	3687	61.83%	4427	74.24%	452	7.58%	452	7.58%	1826	30.62%	893	14.98%	712	11.94%	594	9.96%

TABLE II: Numbers and percentages of encapsulated classes and classes with escaped members

Project	No. of Classes			F/P		F/N	
	Total	Pos.	Neg.	No.	Rate	No.	Rate
ICTCLAS4j	34	12	22	1	8.33%	2	9.09%
Sunflow	183	57	126	6	10.53%	4	3.17%
OC (Memento)	131	24	107	5	20.83%	1	0.93%
OC (NoProxy)	128	22	106	4	18.18%	1	0.94%
OC (OasD)	130	22	108	4	18.18%	1	0.93%
Total	606	137	469	20	14.60%	9	1.92%

TABLE III: False positives and false negatives

effects. To improve pricision, a possible solution would be modeling the behavior of the libraries through their documentation [25]. Then we can get simplified implementations of the methods, based on which we can compute the effects on encapsulation.

REFERENCES

[1] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *European Conference on Object-Oriented Programming*, pages 1–25. Springer, 2004.

[2] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *European Conference on Object-Oriented Programming*, pages 32–59. Springer, 1997.

[3] P. Avgustinov, A. I. Baars, A. S. Henriksen, G. Lavender, G. Menzel, O. de Moor, M. Schäfer, and J. Tibble. Tracking static analysis violations over time to capture developer characteristics. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 437–447. IEEE Press, 2015.

[4] V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 931–940. IEEE Press, 2013.

[5] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 134–144. IEEE, 2015.

[6] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. *Acm Sigplan Notices*, 34(10):1–19, 1999.

[7] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM SIGPLAN Notices*, volume 37, pages 292–310. ACM, 2002.

[8] D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 15–58. Springer, 2013.

[9] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *ACM SIGPLAN Notices*, volume 33, pages 48–64. ACM, 1998.

[10] W. Dietl, S. Drossopoulou, and P. Müller. Generic universe types. In *European Conference on Object-Oriented Programming*, pages 28–53. Springer, 2007.

[11] W. Dietl, S. Drossopoulou, and P. Müller. Separating ownership topology and encapsulation with generic universe types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):20, 2011.

[12] W. Dietl, M. D. Ernst, and P. Müller. Tunable static inference for generic universe types. In *European Conference on Object-Oriented Programming*, pages 333–357. Springer, 2011.

[13] W. M. Dietl. *Universe Types Topology, Encapsulation, Genericity, and Tools*. PhD thesis, Universität Salzburg, 2009.

[14] R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 121–133. ACM, 1998.

[15] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):848–894, 1999.

[16] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *ACM SIGPLAN Notices*, volume 26, pages 271–285. ACM, 1991.

[17] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[18] P. Müller and A. Rudich. Ownership transfer in universe types. In *ACM SIGPLAN Notices*, volume 42, pages 461–478. ACM, 2007.

[19] S. Nägeli. Ownership in design patterns. *Master's thesis, March*, 2006.

[20] J. Noble. Iterators and encapsulation. In *Technology of Object-Oriented Languages, 2000. TOOLS 33. Proceedings. 33rd International Conference on*, pages 431–442. IEEE, 2000.

[21] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *European Conference on Object-Oriented Programming*, pages 158–185. Springer, 1998.

[22] J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness, and immutability. In *International Conference on Objects, Components, Models and Patterns*, pages 178–197. Springer, 2008.

[23] A. Potanin, M. Damitio, and J. Noble. Are your incoming aliases really necessary? counting the cost of object ownership. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 742–751. IEEE Press, 2013.

[24] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 403–414. IEEE Press, 2015.

[25] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin. Automatic model generation from documentation for java api functions. In *Proceedings of the 38th International Conference on Software Engineering*, pages 380–391. ACM, 2016.

[26] T. Zhang, M. Song, J. Pinedo, and M. Kim. Interactive code review for systematic changes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 111–122. IEEE Press, 2015.