ABSTRACT

Software systems use many third party libraries by invoking their APIs. A software system may potentially use an API in an inefficient manner, for example, by creating unnecessary or a large number of short-lived objects. This can cause performance degradation in terms of memory usage and latency in critical applications. In this paper we use an object invocation model based on object creation and their method invocations from different code locations. We use a framework to extract the model features from a running software system. The extracted features are then used in a clustering based mechanism to identify problematic code locations within the software system. We demonstrate our approach by analyzing Java Collection API objects in a Java-based open source editor JEdit. We have successfully identified interesting code locations and discussed their impact on software performance.

1. INTRODUCTION

Developing software systems involves using many third party libraries via their application programming interface (API). To ensure that a software is utilizing an API efficiently, the developer may need to focus on various parameters including the type and number of API objects created, the methods invoked on those objects, and the code locations which created or invoked the objects during their lifetime. Moreover, programmers are interested in knowing these characteristics for identifying different types of runtime behaviors of a given API in their programs. For example, a developer working on the maintenance of a large software system might need to know all the code locations where Java Mail API objects have been used. If all the classes and their method names in the API are well known then a simple Grep like facility [21] can be used to identify such code locations. However, such a trivial approach, would not work properly if 1) the software has defined customized classes extending from the API classes and 2) similar method names are used by any other APIs in the software system. A static code analysis [8, 18, 13] technique can be used to address both of these concerns. But, unfortunately, static code analysis techniques cannot identify the number of objects created and the number of method invocations on those objects, as both of these parameters are dependent upon the runtime behavior of the software system. We propose a dynamic code analysis technique to extract the API usage (in terms of object creation and their method invocations) in any software system. In terms of above example, our proposed scheme will not only highlight the code locations which are using the Java Mail API, but we also identify different runtime behaviors exhibited by those locations in terms of the number of created objects and the methods invoked on those objects. We have applied a clustering technique to identify problematic code locations that could potentially result in performance degradation of the overall software.

This paper makes the following contributions:

- We use API invocation model and the byte-code instrumentation framework for capturing key features of API method invocation patterns in any Java program.
- We apply hierarchical agglomerative clustering on the extracted features to classify source code locations. We then present a mechanism to identify potential performance issues related to API usage and highlight the respective code locations.

This paper is structured as follows. Section 2 describes the API invocation model. We then explain its implementation details including the bytecode instrumentation framework, runtime data extraction and its offline analysis to gather required model parameters in Section 3. In Section 4 we...
2. API INVOCATION MODEL

An API is a set of related classes and their methods that can be used to access the API functionality. The usage of an API within a program can be defined in terms of number and types of objects created, number and types of methods invoked on those objects, and the code locations which created those objects or invoked their methods. Figure 1 shows an abstract API usage scenario depicting four different invocation patterns. In this figure the arrow indicates that a method was invoked on the object from a code location. The locations $i$, $j$, $k$ and $m$ create a number of API objects where each location exhibits a different invocation behavior. All the objects created at location $i$ are invoked numerous times from multiple code locations (i.e., locations 1, 2 and 3). However, the API objects created at location $j$ are invoked only once from a single location. One of the objects created at location $k$ never gets invoked but the other one is invoked multiple times from different code locations, whereas location $m$ has a single method invocation from a single location.

![Figure 1: Objects created at location $i$, $j$, $k$ and $m$ are invoked numerous times from multiple code locations (i.e., locations 1, 2, 3 and 4) exhibiting different invocation behaviors](image)

In this paper, we use the term **API invocation** to mean that a method was invoked on an object belonging to an API. We define the **API invocation model** as follows:

Let $M$ be the set of methods, $C$ be the set of classes and $P$ be the set of packages such that $M = \{m_1, m_2, ..., m_n\}$, $C = \{c_1, c_2, ..., c_y\}$, and $P = \{p_1, p_2, ..., p_z\}$. Moreover if $c_i \in C$ and $c_i \subseteq M$ and $p_i \in P$ then the following conditions hold $m_i \in c_j \wedge m_i \in c_k \Rightarrow (j = k)$ and $c_i \in p_j \wedge c_i \in p_k \Rightarrow (j = k)$.

That is, methods and classes are not shared among classes and packages respectively. We also define **location** as a tuple $loc = (m_j, c_k, p_s)$ where $m_j \in M$, $c_k \in C$, and $p_s \in P$. If $LOC_{instrumented} = \{loc_1, loc_2, ..., loc_m\}$ represents all those locations which need to be instrumented then $O$ represents the set of objects $O = \{o_1, o_2, ..., o_n\}$ s.t. $\forall o_i \in O$, $type(o_i) \in C_B$, $\text{locationCreated}(o_i) \in LOC_{instrumented}$ and $\text{locationsInvoked}(o_i) \in LOC_{instrumented}$, where $C_B \subseteq C$ represents base classes of an API, type($o$) represents the class name of object $o$, locationCreated($o$) represents the location where object $o$ was created and locationsInvoked($o$) represents the set of locations which invoked object $o$ during its lifetime. Notice that we are also interested in those objects which have been destroyed $O_{dest} \subseteq O$.

At any particular location $i \in LOC_{instrumented}$, we define $O_i \subseteq O$ such that $\forall o \in O_i$, locationCreated($o$) = $i$. We also define $O_{dest}^i \subseteq O_{dest}$ such that $\forall o \in O_{dest}^i$, locationCreated($o$) = $i$. Essentially $O_i$ is the set of all those objects which were created at location $i$ and $O_{dest}^i$ is the set of all those objects which were created at location $i$ and then got destroyed afterwards. A method invoked $n$ times on object $o \in O_{dest}^i$ from location $loc$ can be represented as $n = \text{MethodInvoked}(o, loc)$. The set of those objects which were invoked before getting destroyed $O_{invoked}^i$ and the respective set of locations from where these objects were invoked $LOC_{invokes}^i$ can now be calculated as:

$$O_{invoked}^i \subseteq O_{dest}^i \text{ and } LOC_{invokes}^i \subseteq LOC_{instrumented} \text{ s.t. } \text{MethodInvoked}(o, loc) > 0$$

where $o \in O_{dest}^i$ and $loc \in LOC_{instrumented}$.

The clustering features are specified based on the location $i$ which created API objects. In this paper, the features are represented as a tuple $(OC, OU, OI, NI, OIL)$ where $OC = |O_i|$, $OU = |O_{dest}^i|$, $OI = |O_{invoked}^i|$, $NI = \sum_{k=1}^{n} n_k$ and $OIL = |LOC_{invokes}^i|$.

3. IMPLEMENTATION

This section provides the implementation details of our proposed framework. We describe the bytecode instrumentation algorithm which takes any user specified API as input and can be used to instrument any Java program. We also provide the details of runtime data extraction and offline analysis performed on the data obtained from the instrumented program.

3.1 Dynamic Bytecode Instrumentation

The bytecode instrumentation is a well known technique used to inject code into any program. The dynamic bytecode instrumentation is generally used to inject any code into a program at runtime during its execution. The dynamic bytecode instrumentation technique has been widely used for fault detection, performance monitoring, and debugging.

Java provides a standard library `java.lang.instrument` which can be used to get access to the Java classes when they are loaded during program execution. The actual instrumentation (i.e., to identify the required location and to inject the code) can be performed using any one of the available instrumentation libraries like BCEL [6], Javassist [10], ASM [3] etc. We have used ASM [3] for bytecode engineering.

In order to collect required API specific object invocation information, our byte-code instrumentation technique had to address the following questions:
- Which packages or modules should be instrumented: The user had to specify the list of classes which should be instrumented using a regular expression. A regular expression like org/gjt/sp/jedit.* means that all the classes in the package org/gjt/sp/jedit and all its sub-packages must be instrumented.

- Which API objects are being considered for invocation analysis: The user had to specify the list of API classes whose objects were considered for invocation analysis. For example, Java Collection API could be defined as java.util.Collection|java.util.Map which means that any instance of these or any of their sub-classes are considered to be a Java Collection API object.

- How to identify and instrument the code locations which create new objects (belonging to user specified API) or invoke a method on them: This is addressed by our bytecode instrumentation algorithm shown in Figure 2. Java instrumentation library (java.lang.instrument package) enables us to access every class which is loaded by the Java Virtual Machine (JVM) for the instrumented program. The algorithm first determines if the class being loaded belongs to the set of classes C_t which need to be instrumented. If it does, then it goes through every method in the class and finds every instruction which creates a new object. It identifies the type t_o of the object and if the type inherits from any of the user specified API classes then it inserts a utility function named objectCreated(o) after the first constructor call. The algorithm then searches every method in the class being loaded for a method invocation. It identifies the type of the object invoking the method. Again, if the type of the object inherits from any of the user specified API classes then it inserts a utility function named objectInvoked(o) right after the method invocation.

- How to extract the required object invocation information at runtime: This process is explained in the next subsection.

3.2 Runtime Data Extraction and Analysis
As described in the algorithm in Figure 2, the bytecode instrumentation will insert methods objectCreated(o) and objectInvoked(o) in the instrumented code to identify the API specific object creation and object usage information respectively.

The objectCreated(o) method will be invoked every time a new object belonging to the API being analyzed is created. Each object is assigned a unique identifier and its runtime type and creation location (i.e. the class/method which created this object) is stored in a comma separated log file named ObjectCreations.log.

Similarly, objectInvoked(o) will be called whenever an object belonging to the API is invoked from the instrumented code. The object access information which include object’s unique id and its accessor location (i.e. the class/method which invoked method on this object) is stored in a comma separated log file named ObjectUsage.log.

Input: C_t \subset C, A set of classes which need to be instrumented.
Input: C_B \subset C, A set of base classes belonging to an API.
Input: M_B, A set of method signatures such that \forall m \in C_B, M_B = \bigcup m_o and \forall m_o \in c_o, m_o was invoked from a class c_o \in C_t

1: for all c \in C_t do
2: for all m \in c do
3: Identify code location in m where a new object o has been created
4: Identify the type t_o of the object o
5: if t_o extends from a class c_o \in C_B then
6: Identify the location L_new, where the first constructor call has been made for object o
7: Insert the code ObjectCreated(o) at code location L_new
8: end if
9: Identify code location in m where a method m_i has been invoked on an object x
10: Identify the type t_x of the object x on which the method has been invoked
11: if t_x extends from a class c_i \in C_B then
12: if m_i \in M_B then
13: Identify the location L_invoked, where the object x has been invoked
14: Insert the code objectInvoked(x) at code location L_invoked
15: end if
16: end if
17: end for
18: end for

Figure 2: Bytecode Instrumentation Algorithm

In order to be precise in our analysis, we had to consider only those objects which had completed their lifecycle (i.e. which had been destroyed). However, identifying object destruction was tricky in Java, since there is no explicit code statement to destroy objects. The objects are garbage collected automatically by a background thread. We had to use a technique based on Phantom references to keep track of the object destruction. A method named objectDestroyed(o) was invoked every time an object was destroyed by the garbage collector. The object destruction information including the object’s unique identifier was also stored in a comma separated log file named ObjectDestroyed.log.

The cluster analysis was then performed using the log files generated during the runtime data extraction. At first, ObjectCreations.log and ObjectDestroyed.log files were processed to identify the set of objects which had completed their life during program execution trace. The ObjectUsage.log was then processed to gather required clustering features described in Section 2. Since all the features are numeric, we chose to use Hierarchical Agglomerative Clustering using Euclidean distance function. The features were normalized using Min-Max normalization technique. The results are presented in the next section.

4. EXPERIMENTAL RESULTS
We performed our experimental study on a Java based open source text editor jEdit. It is a large open source project consisting of about 500 Java source code files (with 115000 lines of code). We chose to instrument this program to extract and analyze invocations on Java Collection API objects. Any object which inherited from java.util.Collection or java.util.Map was considered as Java Collection API object.

The execution trace was obtained by executing the instrumented jEdit program for approximately thirty (30) minutes. The execution trace included data for jEdit startup, creating new files, opening existing files, saving modified files, finding and replacing keywords, hyper-search and plugins management related operations. The execution trace consisted of about 6 million log entries which were created pertaining to object creation, method invocations and destructions. During the program execution 14104 Java Collection API objects were created, out of which 13527 objects were destroyed within the execution trace time period. The 9134 objects out of 13527 were invoked 176685 times from 214 different code locations. All of these objects were created from 153 different code locations.

The execution trace was then processed and the required features were extracted for each code location which had created any Java Collection API object. As described in Section 2, the extracted feature set includes number of objects created (OC), number of objects destroyed (OU), number of objects invoked (OI), number of total invocations (NI) and number of different code locations invoking the objects (OIL). Figure 3 shows some of the sample features extracted for Java Collection API usage in jEdit program.

We then used RapidMiner [14] to apply hierarchical agglomerative clustering using Average Linkage rule on the extracted features to discover the complete hierarchy of clusters. The clustering was based on Euclidean distance function, as all the features were numeric. The input data for the clustering was normalized using standard min-max normalization technique. We then used cluster flattening to flatten the resulting cluster hierarchy at 6th level, and thus identifying six (6) high level clusters. The resulting clusters are shown in Figure 4 and the respective code locations are shown in Figure 5. Please note that the normalized values of the features plotted on the y-axis are in the range [0 – 1]. Each cluster provides a unique perspective on the API invocation behavior. In the following we will discuss the characteristics of these clusters, highlighting their significance for performance improvements.

- **cluster_4** provides the most glaring performance problem. It identifies two source code locations which create quite large number of API objects which almost never get invoked during their lifetime. The developer can potentially optimize the code to avoid unnecessary object creation at these code locations.

- **cluster_2** and **cluster_3** consist of those code locations which are creating excessive number of short-lived objects. Although all of these objects get invoked, but these objects get discarded after few invocations. Figure 5 shows the respective source code locations and the developer can investigate them further to determine if the code can be optimized to work with fewer number of objects.

- **cluster_0, cluster_1** and **cluster_5** do not indicate any performance concerns. **cluster_0** identifies that most of the code locations are creating a few number of API objects which are discarded after few invocations. **cluster_1** identifies two code locations which create relatively fewer number of objects but these objects are excessively invoked during their lifetime. **cluster_5** identifies a code location which creates relatively fewer number of objects, but these objects are invoked from many different code locations. A closer look at the jEdit source code shows that all these invocations are made from different methods of the same JEditBuffer class. It would have been a serious design concern if these invocations were from different classes or packages.

5. RELATED WORK

This work is an extension of our earlier work [12] that was based on identifying source code locations using object invocation ratio and object invocation frequency. In this paper, we have classified source code locations by applying a clustering technique on extracted features. We use these classifications to identify performance issues.

There has been substantial amount of research work conducted on inferring API properties from existing code base as summarized in [17]. As highlighted by this paper, much of the existing work on API usage analysis is focused on extracting API documentation and identifying frequent usage patterns. These works can be broadly categorized into static [24, 8, 19, 1] and dynamic [22, 0] approaches. A large focus of the API usage research has been directed towards static code analysis. [24] extracts recurring patterns of given API element (i.e. an API class and/or method) and applies sequence mining and clustering to group similar patterns together. It then recommends useful code snippets to the users based on these mined patterns. [19] performs static code analysis to identify API usage concepts from version histories. [22] uses dynamic analysis to determine API usage patterns from imperfect traces as partial orders.

Our approach resembles some of the general approaches which are based on API usage statistics and that focus on understanding API coverage within a code base [8, 15]. The key difference is that our approach is based on dynamic program...
### Figure 3: Sample features used for clustering Java Collection API usage in jEdit program

<table>
<thead>
<tr>
<th>Code Locations</th>
<th>OC</th>
<th>OU</th>
<th>OI</th>
<th>NI</th>
<th>OIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.gjt.sp.jedit.jEdit.openFile</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>41</td>
<td>2</td>
</tr>
<tr>
<td>org.gjt.sp.jedit.bsh.CallStack.&lt;init&gt;</td>
<td>654</td>
<td>653</td>
<td>501</td>
<td>4740</td>
<td>6</td>
</tr>
<tr>
<td>org.gjt.sp.jedit.bsh.NameSpace.setMethod</td>
<td>69</td>
<td>50</td>
<td>50</td>
<td>150</td>
<td>2</td>
</tr>
<tr>
<td>org.gjt.sp.jedit.gui.DockableWindowManager$KeyHandler.parseShortcut</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>org.gjt.sp.jedit.io.AutoDetection.getEncodingDetectors</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>20</td>
<td>1</td>
</tr>
</tbody>
</table>

### Figure 5: Clustering results identifying different types of code locations for jEdit program

<table>
<thead>
<tr>
<th>Code Locations</th>
<th>OC</th>
<th>OU</th>
<th>OI</th>
<th>NI</th>
<th>OIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>cluster_0 (141 locations)</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>cluster_1 (2 locations)</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>32041</td>
<td>2</td>
</tr>
<tr>
<td>cluster_2 (2 locations)</td>
<td>1319</td>
<td>1319</td>
<td>1319</td>
<td>7804</td>
<td>2</td>
</tr>
<tr>
<td>cluster_3 (1 location)</td>
<td>2767</td>
<td>2767</td>
<td>2767</td>
<td>8301</td>
<td>2</td>
</tr>
<tr>
<td>cluster_4 (2 locations)</td>
<td>1452</td>
<td>1450</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cluster_5 (1 location)</td>
<td>972</td>
<td>968</td>
<td>2</td>
<td>27</td>
<td>2</td>
</tr>
</tbody>
</table>

The runtime objects based models have been extensively used in the dynamic program analysis literature for program comprehension, anomaly detection, and to identify performance problems. Our approach targets gaining a deeper insight into API usage within a single project based on its runtime characteristics. Yourkit, which is a leading application performance measurement tool, also provides the statistics on the total number of objects created or referenced within a program. However, it does not classify the source code locations based on the method invocation behaviors of those objects.

The runtime objects based models have been extensively used in the dynamic program analysis literature for program comprehension, anomaly detection, and to identify performance problems. Our approach targets gaining a deeper insight into API usage within a single project based on its runtime characteristics. Yourkit, which is a leading application performance measurement tool, also provides the statistics on the total number of objects created or referenced within a program. However, it does not classify the source code locations based on the method invocation behaviors of those objects.

In general, the existing approaches focus on extracting program or API specifications, identifying common patterns of object usage model and visualizing the runtime object interactions. Our approach differs in that it focuses on API usage based on method invocations (rather than class/method references) and identifies the source code locations which exhibit certain method invocation behavior of API objects.
6. CONCLUSION
We have presented a mechanism to classify source code locations in any Java program based on method invocation patterns of a user defined API. Clustering is used for classification where a feature set is defined using API invocation model and the features are extracted using byte-code instrumentation framework. The classification helps the programmers in identifying several potential performance issues in the code base. We demonstrate the efficacy of our proposed framework by performing clustering based invocation analysis of Java Collection API objects in a large open source program jEdit. The resulting clusters help the programmers in identifying code locations which are creating unnecessary objects or a large number of short lived objects. The programmers can use this information to improve the memory usage and latency of their respective applications.

7. ACKNOWLEDGMENTS
This work was partially supported by Prince Mohammad Bin Fahd University. The views and conclusions herein are those of the authors and do not represent the official policies of the university.

8. REFERENCES