SpotWeb: Characterizing Framework API Usages Through a Code Search Engine

Suresh Thummalapenta[1] and Tao Xie[2]

Department of Computer Science,
North Carolina State University, Raleigh, USA.
[1]sthumma@ncsu.edu,[2]xie@csc.ncsu.edu**

Abstract. The essentials of modern software development (such as low cost and high efficiency) demand software developers to make intensive reuse of the existing open source frameworks or libraries (generally referred as frameworks) available on the web. However, developers often face challenges in reusing these frameworks due to several factors such as the complexity and lack of proper documentation. In this paper, we propose a code-search-engine-based approach that tries to detect *hotspots* in a given framework; these hotspots are the APIs that are frequently reused. Hotspots can serve as starting points for developers in understanding and reusing the given framework. Our approach also detects *deadspots*, which are the APIs that are rarely used. Deadspots serve as caveats for developers as there can be difficulties in finding related code examples and are generally less exercised compared to hotspots. We developed a tool, called SpotWeb, for frameworks or libraries written in the Java programming language and used our tool to detect hotspots and deadspots of eight open source frameworks including JUnit, Log4j, Grappa, JGraphT, OpenJGraph, JUNG, BCEL, and Javassit.

1 Introduction

Reuse of existing open source frameworks or libraries (referred as frameworks) has become a common practice in the current software development process due to several factors such as low cost and high efficiency. However, existing frameworks or libraries often offer complex procedures that may also involve call backs (such as GUI libraries), making these procedures complex for effective reuse. This complexity also makes the documentation of the framework a vital resource. However, the documentation is often missing for many existing frameworks and even if such documentation exists, it is often outdated [9].

In general, frameworks expose certain areas (APIs) of flexibility that are intended for reuse by their users. Software developers who reuse APIs of these frameworks must be aware of these flexible areas for effective reuse of frameworks. These areas of flexibility are often referred as *hotspots*. As described by Pree [12]

** This work is supported in part by NSF grant CNS-0720641 and Army Research Office grant W911NF-07-1-0431.

and Flores et al. [2], hotspots depict a framework's flexibility and proneness to reuse. The foundations of hotspots are built upon the Open-Closed principle by Martin [10]. The Open-Closed principle encompasses two main definitions: the "closed" and the "open" parts. The "open" parts represent areas that are flexible and variant, whereas the "closed" parts represent areas that are immutable in the given framework. The "closed" parts are commonly referred as "templates" and the "open" parts are referred as "hooks" as the users can redefine the behavior of these parts. The combination of templates and hooks, referred as *hotspots*, supports adherence to the Open-Closed principle of Martin.

Hotspots are useful to both users and developers of the framework in several ways. First, new users can browse and inspect hotspots to understand commonly reused APIs and find out the APIs that the users want to reuse. Second, users may have more confidence or tendencies in reusing hotspots because generally bugs in these hotspots may be fewer (or more easily exposed previously) than the ones in non-hotspots; we can view the code reusing hotspots to be a special type of test code that can help expose bugs in hotspots. Third, developers or maintainers of these frameworks can choose to invest their improvement efforts (e.g., performance or quality improvement) on these hotspots because the resulting returns on investment may be substantial.

In contrast to hotspots, we call a framework's areas that are rarely used by users as *deadspots*. The concept of deadspots is introduced by our approach and these deadspots can serve as caveats to users of the given framework. As deadspots represent the rarely used APIs, there can be difficulties in identifying related code examples that can help users in reusing those APIs. Moreover, deadspots are generally less tested compared to hotspots with regards to the "testing" conducted by API client code as test code.

Due to lack of proper documentation, detecting hotspots from the existing open source frameworks or libraries has become an interesting research area and is addressed by several previous approaches [1–3, 5, 11]. However, these approaches either require additional efforts from developers of those frameworks or solely rely on the knowledge available within the framework. In this paper, we propose an approach, called SpotWeb, to address these two preceding problems by exploiting a code search engine (CSE) and without requiring any additional efforts from the framework developers. Detecting hotspots of a framework requires domain knowledge of how the APIs of the input framework are reused by applications, referred as client applications. Furthermore, the effectiveness of hotspot detection mainly depends on the number of client applications used for detecting hotspots. The rationale behind this assumption is that a higher number of client applications can often help in detecting hotspots more effectively. Therefore, our approach uses a CSE such as Google [4], Koders [7], and Krugle [8] that can search in billions of lines of open source code available on the web and uses gathered code samples for detecting hotspots of the input framework. SpotWeb is the first approach that extends the scope of client application code bases for detecting hotspots to billions of lines of open source code by leveraging a code search engine.

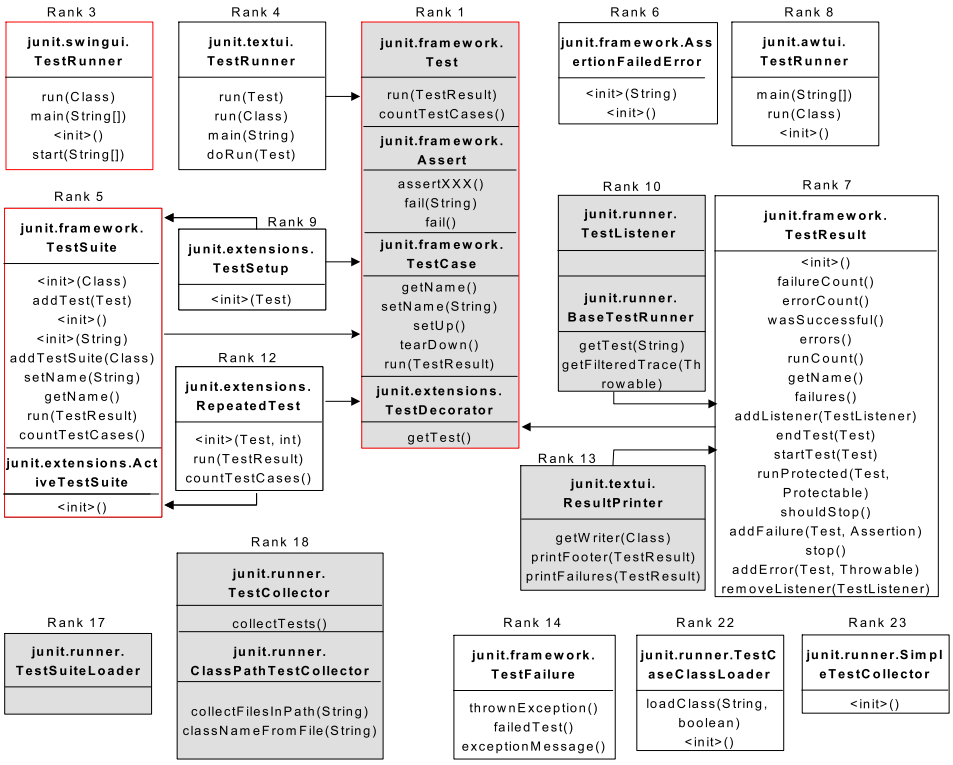

Fig. 1. Hotspots identified in the JUnit framework

In particular, SpotWeb accepts a framework as input and extracts the API information of the input framework. SpotWeb interacts with a CSE to gather relevant code samples for all classes and interfaces of the input framework. SpotWeb analyzes these code samples statically and computes a heuristic called *UsageMetric* that quantitatively evaluates APIs of the input framework. The computed *UsageMetrics* capture metrics related to classes, interfaces, and methods of the input framework. For example, the *UsageMetrics* for a class capture how often the class is instantiated or extended by the client applications. SpotWeb structurally propagates the computed information through five phases for detecting hotspots of the input framework. For each hotspot, SpotWeb also gathers relavant code examples that can assist the framework users in identifying how to reuse the detected hotspots.

We used SpotWeb to identify hotspots and deadspots of eight widely used open source frameworks or libraries including JUnit, Log4j, Grappa, JGraphT, OpenJGraph, JUNG, BCEL, and Javassit, which differ in size and application purpose. We show that SpotWeb can give a high recall while detecting hotspots and the detected hotspots of Log4j and JUnit are consistent with the starting points described in the documentation for these frameworks.

The rest of the paper is organized as follows. Section 2 explains our approach through an illustrative example. Section 3 presents related work. Section 4 de-

```
01:     public class ShipReleaseDAOTestCase extends TestCase {
02:        private ShipReleaseDAO dao = null; ...
03:        public ShipReleaseDAOTestCase() { super();  ...  }
04:        protected void setUp() throws Exception {   ...
05:           dao = (ShipReleaseDAO)context.getBean("shipReleaseDAO"); ...}
06:        public void tearDown() throws Exception { dao = null; }
07:        public void testF() { ... }
08:        public void testB() { ... }
09:     ...}
```

Fig. 2. Suggested code example for the hook class `TestCase`.

```
01:     public class MyTestSuite { ...
02:        public static Test suite(){
03:           TestSuite suite = new TestSuite("All axis.soap tests");
04:           suite.addTest(new ShipReleaseDAOTestCase());
05:           return suite;     }
06:     ...}
```

Fig. 3. Suggested code example for the template class `TestSuite`.

scribes our approach. Section 5 discusses evaluation results. Section 6 discusses threats to validity. Finally, Section 7 concludes.

2 Example

We next use an example to explain our approach and show how the detected hotspots and deadspots can be used by the framework users. We use JUnit [6], the *de facto* standard unit testing framework for the Java programming language, as an illustrative example for explaining our approach.

SpotWeb accepts an input framework, say JUnit, and extracts *ApplicationInfo* from the framework. The *ApplicationInfo* includes all classes, all interfaces, public or protected methods of each class and interface, and inheritance hierarchy among classes or interfaces of the framework. SpotWeb constructs different queries for each class or interface and interacts with a CSE such as Google Code Search [4] to gather relevant code samples from existing open source projects that use the APIs of the input framework. For example, SpotWeb constructs a query such as "`lang:java junit.framework.TestSuite`" for gathering related code samples of the `TestSuite` class. The gathered code samples are referred as a *LocalRepository* for the input framework. SpotWeb analyzes the gathered code samples statically and computes *UsageMetrics* for classes, interfaces, and public or protected methods of all classes and interfaces. For example, the *UsageMetrics* computed for the `TestSuite` class show that the class is instantiated for 165 times and is extended for 32 times. Similarly, the *UsageMetrics* computed for the method `addTest` of the `TestSuite` class show that the method is invoked for 95 times. SpotWeb also gathers code examples for each class or method and stores these code examples in a repository, referred as *ExampleDB*. Then SpotWeb uses the algorithm shown in Figure 6 for detecting hotspots from the computed *UsageMetrics*.

Initially, SpotWeb ranks methods in non-ascending order based on their *UsageMetrics* and uses a threshold percentage t to detect hotspot methods: the methods in the top t percentage with a non-zero *UsageMetrics* are detected

as hotspot methods. The detected hotspot methods are then clustered into their declaring classes, detected as hotspot classes. These hotspot classes are ranked based on the minimum rank of the hotspot methods declared by these classes. SpotWeb classifies the hotspot classes into two categories (templates and hooks) based on heuristics described in the algorithm. The hotspot classes of each category are further clustered into hierarchies based on their inheritance relationships. For example, SpotWeb detected classes `Assert` and `TestCase` as hook hotspots in the JUnit framework. As `TestCase` class extends `Assert` class, SpotWeb clusters both the classes into the same hierarchy. SpotWeb assigns a rank to each hierarchy based on the minimum rank of the hotspot classes contained in the hierarchy. For example, consider that the `Assert` class has Rank 3 and the `TestCase` class has Rank 6, then the clustered hierarchy of the `Assert` and `TestCase` classes is assigned with Rank 3. The rank attribute uniquely identifies a hierarchy among all other hierarchies. Hierarchies with lower ranks have more preference or importance to the hierarchies with higher ranks.

Figure 1 shows the hotspot hierarchies detected in the JUnit framework. The hierarchies that are referred in the succeeding description are highlighted with a red border. The figure also shows ranks assigned to each hierarchy. As the rank attribute uniquely identifies a hierarchy, we use the rank as an identity for describing a hierarchy. Each hierarchy includes one or more hotspot classes and is shown as pairs of class and its methods. For example, Hierarchy 1 (hierarchy with Rank 1) has classes `Test`, `Assert`, `TestCase`, and `TestDecorator`. Hierarchy 1 also shows that the class `Test` includes hotspot methods `run` and `countTestCases`. We show template hierarchies in white and hook hierarchies in gray. For example, Hierarchy 1 is a hook hierarchy and Hierarchy 3 is a template hierarchy.

Methods inside each class of each hierarchy are sorted based on their computed *UsageMetrics*. Sorting of methods of a class can assist the framework users in quickly identifying the methods that are often used inside a given hotspot class. For example, consider the `TestSuite` class shown in Hierarchy 5. The `TestSuite` class has three constructors `<init>(Class)`, `<init>()`, and `<init>(String)`. However, the constructor `<init>(Class)` is often used compared to the other two constructors. In the class `Assert` of Hierarchy 1, the method name `assertXXX` indicates different assertion methods such as `assertEquals` and `assertTrue` of the class `Assert`.

The figure also displays dependencies among hotspot hierarchies (shown as arrows between hierarchies). SpotWeb tries to capture the usage relationships among hotspot classes through dependencies. For example, if a template class, say X, has a constructor that requires an instance of another template class, say Y, then SpotWeb captures dependency of the form "X $\rightarrow$ Y", which describes that X requires Y. Basically, SpotWeb identifies two kinds of dependencies: `TEMPLATE_TEMPLATE` and `TEMPLATE_HOOK`. Each dependency has a parent and a child, and the dependency relation is shown from the child to the parent. A `TEMPLATE_TEMPLATE` dependency indicates that an instance of the parent template hotspot is required for using the child template hotspot. A `TEMPLATE_HOOK` dependency describes that the user has to define a new behavior for the corresponding

hook hotspot before using the template hotspot. For example, Hierarchy 5 has a `TEMPLATE_HOOK` dependency with Hierarchy 1. This dependency indicates that to reuse methods such as `addTest` of the class `TestSuite` in Hierarchy 5, the user has to define a new behavior for the classes in Hierarchy 1.

We next describe how the hotspots detected by SpotWeb can be used by the framework users to reuse the APIs of the JUnit framework. After reviewing the hotspots shown in Figure 1, consider that a framework user wants to start with the method `addTest` of the template class `TestSuite` in Hierarchy 5. Figure 1 shows that Hierarchy 5 of the `TestSuite` class has a `TEMPLATE_HOOK` dependency with the Hierarchy 1. This dependency indicates that the user may need to define a new behavior for the associated hook hierarchy. SpotWeb recommends the code example shown in Figure 2 for the hook class `TestCase`, which is a part of Hierarchy 1. The code example exhibits several aspects that needs to be handled by the user while extending the `TestCase` class. For example, in the `SetUp` method the user can write code for setting up the environment such as instantiating necessary variables, and in the `tearDown` method the user can destroy the created variables. Also, the code example shows that names of the test methods in the extended class of the `TestCase` should start with the prefix `test`. SpotWeb also recommends code example for `addTest` method and the recommended code example is shown in Figure 3. The code example shows that the user has to create an instance of the `TestSuite` class and then add test cases through the `addTest` method.

An API is identified as a deadspot if that API is neither used directly nor used indirectly by the gathered code samples. The complete algorithm used for detecting deadspots is shown in Figure 7. SpotWeb identified 20 classes such as `Swapper`, `TestRunListener`, and `ExceptionTestCase` as deadspots in the JUnit framework. However, deadspots are only suggestions for users unfamiliar to that framework or library and SpotWeb does not intend to recommend users not to reuse those deadspot classes. Sometimes, deadspots can also be helpful to the framework developers in distributing their maintenance efforts, because the framework developers can give a low preference to the deadspot classes.

3 Related Work

Detecting hotspots in a given framework is an interesting area of research for many years and is addressed by several previous approaches. However, SpotWeb is the first approach that leverages a code search engine for gathering the knowledge of how the input framework APIs are reused by other client applications. As SpotWeb uses a CSE, SpotWeb can mine a much larger scope of code bases compared to other approaches.

Hotspotter by Flores et al. [2] is closely related to our SpotWeb. Hotspotter identifies hotspots in the given library or framework through a JavaML base representation of the source code and evolves through a series of XSL transformations. SpotWeb is different from Hotspotter as Hotspotter identifies hotspots with only the knowledge available within the input framework. Instead, SpotWeb identifies hotspots from the knowledge both within the input application and

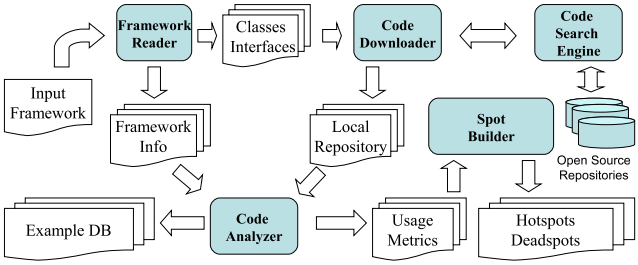

Fig. 4. Overview of SpotWeb approach

from code samples gathered from a CSE. Therefore, the results of SpotWeb can be more precise compared to the results of Hotspotter.

Baxter et al. [1] proposed an approach to discover the structure of Java programs and the way that the classes relate to each other through inheritance and composition. Their study is useful for the framework developers who can evaluate the structural features of their own programming practice and optimize their performance. Instead, SpotWeb is useful for the framework users in effectively reusing the APIs of the framework.

Mendonca et al. [11] proposed an approach to assist framework instantiation and to understand the intricate details surrounding the framework design. However, their approach requires users to have knowledge regarding a specific process language, called Reuse Definition Language, proposed by their approach. Other several previous approaches [3, 5] also claim to facilitate framework instantiation. However, these approaches need an additional effort from the framework developers. Unlike these approaches, SpotWeb does not need any additional effort from the framework developers. Our approach tries to gather the required additional information from code samples gathered through a CSE.

Our previous approaches MAPO [14] and PARSEWeb [13] also exploit CSEs for gathering related code examples. However, these previous approaches are developed for assisting the users in effectively reusing a given API. Instead, SpotWeb assists the framework users by detecting hotspots that can serve as starting points for reusing the framework.

4 Approach

Our approach consists of five major components: the framework reader, code search engine (CSE), code downloader, code analyzer, and spot builder. Figure 4 shows an overview of all components and flows among different components. The framework reader component takes a framework as input and extracts the *FrameworkInfo* information. The code downloader accepts a set of classes and interfaces from the framework reader as input and interacts with a CSE to download relevant code samples. The downloaded code samples, referred as *LocalRepository*, are given as input to the code analyzer. The code analyzer analyzes code samples stored in the *LocalRepository* statically and computes *UsageMetrics* for all classes and methods of the input framework. The spot builder component uses the computed *UsageMetrics* for detecting hotspots and deadspots. The code analyzer also identifies several code examples for each class and method, and stores these code examples in a repository, referred as *ExampleDB*.

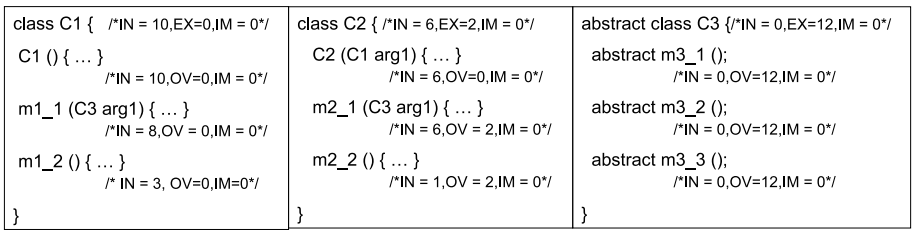

Fig. 5. Example classes of a sample input framework

4.1 Framework Reader

The framework reader component accepts the input framework and gathers entire API information, referred as *FrameworkInfo*. The *FrameworkInfo* includes the set of package names, all classes, all interfaces, and public or protected methods of all classes and interfaces. The framework reader also gathers the inter-method calls of the input framework; these inter-method calls give the set of other public and protected methods invoked by each method of the class and are used while identifying deadspots. The framework reader also extracts inheritance hierarchy among classes or interfaces of the input framework.

4.2 Code Search Engine

Code Search Engines (CSE)[1] are primarily used by programmers to search for relevant code samples. As CSE can search in billions of lines of open source code available on the web, CSE can serve as powerful resources of open source code. Therefore, we used a CSE in our approach to gather relevant code samples of the given framework. In our approach, we used Google Code Search (GCS) [4] for collecting relevant code samples, partly because GCS provides convenient open APIs for third-party tools to interact with and it has been consistently improved and maintained. However, our approach is independent of the underlying CSE and can be extended easily to any other CSE.

4.3 Code Downloader

The code downloader accepts a set of classes and interfaces as input from the framework reader component. The code downloader constructs different queries for each element in the set and interacts with the CSE to gather relevant code samples. For example, the code downloader constructs the query "`lang:java junit.framework.TestSuite`" to collect code samples related to the `TestSuite` class of the JUnit framework. The gathered code samples are stored in a *LocalRepository* and serve as input to the code analyzer component. Along with code samples, the code downloader also stores the project information associated with each code sample.

4.4 Code Analyzer

The code analyzer accepts the *FrameworkInfo* from the framework reader and the *LocalRepository* from the code downloader as inputs, and analyzes the gathered code samples stored in the *LocalRepository* statically to compute *UsageMetrics* for all classes and methods of the input framework.

[1] `http://gonzui.sourceforge.net/links.html`

The *UsageMetrics* capture several ways of how often each class or interface or a method of the input framework is used by the gathered code samples. For example, a class can be instantiated to invoke its methods or the class can be extended to define a new behavior. Similarly, a method of a class can be either invoked or overridden. Furthermore, some methods such as constructors or factory methods may be invoked only once in a client application and other methods may be invoked several times. For example, in the JUnit framework, the `TestSuite` object may be created once but the `addTest` method may be invoked several times for adding test cases to the test suite. Therefore, computing metrics without considering the associated project information can result in a higher biased preference to the `addTest` method instead of the actual starting point, which is the constructor call of the `TestSuite` class. Therefore, the code analyzer computes the usage for a class or a method only once for a client application. For example, consider that the `addTest` method is used 20 times by a client application. But the code analyzer considers that the `addTest` method is used only once by the client application.

The *UsageMetrics* for a class include the number of created instances (more precisely, the number of constructor call sites) and the number of times that the class is extended. For an interface, the *UsageMetrics* include the number of times that the interface is implemented. We use notations IN_j, EX_j, and IM_j for the number of instances, the number of extensions, and the number of implementations, respectively. The consolidated usage metric UM_j for a class or an interface is the sum of all the three preceding metrics. The code analyzer computes three types of *UsageMetrics* for methods: Invocations, Overrides, and Implements. The *Invocations* metric gives the number of times that the method is invoked by the code samples. The *Overrides* metric gives the number of times that the method is overridden by the code samples to define a new behavior. The *Implements* metric, specific for interfaces, gives the number of times that the method is implemented. For constructors, the code analyzer computes only the *Invocations* metric. We use notations IN_i, OV_i, and IM_i for invocations, overrides, and implementations, respectively. The overall usage metric for a method is the sum of all the three preceding metrics.

The code analyzer also gathers code examples for each class or method and stores these code examples in a repository, referred as `ExampleDB`. The `ExampleDB` is used for suggesting related code examples for a class or a method requested by the user. The related code examples can further assist users in making an effective reuse of APIs of the input framework.

4.5 Spot Builder

The spot builder component (SBC) accepts the computed *UsageMetrics* and detects hotspots and deadspots by structurally propagating the computed information through five phases. We next describe how SBC identifies hotspots and deadspots.

Identification of hotspots: Hotspots are APIs that are often used by the gathered code samples stored in the *LocalRepository*. The algorithm used by SBC for detecting hotspots is shown in Figure 6. We next describe the algorithm

```
Input: UsageMetrics of classes and methods, HT percentage
Output: Hotspot hierarchies and their dependencies
SortedMET = Sort methods based on their usage metric values;
for (MET_i in SortedMET) {
    if(UM_i ≠ 0) {
        if(Position of MET_i ≤ (HT * Size of SortedMET)){
            Set MET_i type as HOTSPOT;}
        else {
            Set MET_i type as WEAKSPOT;}
}
CT_j = Cluster HOTSPOT MET_i into related classes;
Rank of CT_j = Minimum rank of all MET_i of the CT_j;
//Classify CT_j into templates and hooks
Switch(CT_j) {
    Case (Interface): Set CT_j type to HOOK;
    Case (Abstract Class): Set CT_j type to HOOK;
    Other :{
        if(EX_j of CT_j > IM_j of CT_j) {
            Set CT_j type to HOOK;}
        else {Set CT_j type to TEMPLATE;}}
Cluster CT_j of the same category into hierarchies based on inheritance;
Associate hook hierarchies to template hierarchies;
Define dependencies between template hierarchies;
```

Fig. 6. Algorithm for detecting hotspots through the computed UsageMetrics

through an illustrative example shown in Figure 5. The figure shows three classes C_1, C_2, and C_3 and their declarations. The class C_3 is an `abstract` class. The figure also shows the computed usage metrics for each class and its methods. For example, the class C_1 is instantiated for 10 times (shown as `IN`=10) and the abstract class C_3 is extended for 12 times (shown as `EX`=12). Similarly, the method m_{2_1} is invoked for 6 times and is overridden for 2 times.

Initially, SBC sorts all methods such as m_{1_1} and m_{2_1} based on their computed usage metric values. SBC uses a threshold percentage, referred as HT, and selects the top HT methods, whose usage metric is non-zero, as hotspots. For example, for a HT of 7%, SBC identifies the methods as m_{3_1}, m_{3_2}, m_{3_3}, c_1, and so on as hotspot methods. SBC traverses the hotspot methods and clusters them into their declaring classes. The clustered classes are sorted based on the minimum rank among their methods. In the current example, the clustering process results in classes C_3 (methods: m_{3_1}, m_{3_2}, and m_{3_3}), C_1 (methods: c_1 and m_{1_1}), and C_2 (methods: c_2 and m_{2_1}). After clustering, SBC uses the computed metrics of classes to classify these classes further into templates and hooks. The criteria used for classifying hotspot classes into templates and hooks are shown in the algorithm. For the current example, SBC identifies class C_3 as a `HOOK` class and classes C_1 and C_2 as `TEMPLATE` classes. SBC further tries to cluster the classes of the same category based on their inheritance relationship. For example, if C_1 has a parent class P_1 and both the classes are classified as `TEMPLATE` classes, SBC clusters C_1 and P_1 into the same hierarchy.

```
Input: A method M_i of a class C_j
Output: Is the method a dead spot or not?
if (UM_i ≠ 0) {Return false;}
if (C_j is an interface) {
      //verify all implemented methods of M_i
      if (All implemented methods of M_i are deadspots) { Return true; }
          else { Return false; }}
if (M_i is abstract) {
      //Verify all overridden methods of M_i
      if (All overridden methods of M_i are deadspots) {Return true;}
          else { Return false; }}
if (All callers of M_i are deadspots) { Return true; } else { Return false; }
```

Fig. 7. Algorithm for detecting whether a method is a deadspot.

SBC identifies dependencies among the detected hotspot hierarchies based on the arguments passed to methods of the classes. SBC identifies two kinds of dependencies: `TEMPLATE_HOOK` and `TEMPLATE_TEMPLATE`. A `TEMPLATE_HOOK` dependency defines a relationship between a template hierarchy and a hook hierarchy. SBC identifies that a template hierarchy is dependent on a hook hierarchy if methods in the template hierarchy accept the classes of the hook hierarchy as arguments. Such a dependency describes that the users have to first define a new behavior for those related hook classes, say extend the class, and use the instances of those classes as arguments. In the given code example, SBC identifies that class C_1 has a `TEMPLATE_HOOK` dependency with the class C_3 as the method m_{1_1} requires an instance of C_3 as an argument. Similarly, the spot builder identifies `TEMPLATE_TEMPLATE` hierarchies, when a template hierarchy is dependent on another template hierarchy. For example, class C_2 has a `TEMPLATE_TEMPLATE` dependency with the class C_1. In SpotWeb implementation, we used the HT percentage as 25%, which is based on our evaluations with frameworks JUnit and Log4j.

Identification of deadspots: SBC identifies APIs of the input framework that are rarely or never used by the gathered code samples as deadspots. However, detecting deadspots based on only the *UsageMetrics* can give many false positives. For example, the *UsageMetrics* for an abstract method defined in a class can be zero, as the gathered code samples reference the concrete implementation provided by some of the classes's subclasses. In this case, this abstract method is not a deadspot as the method is indirectly referenced through the subclasses. Therefore, to reduce the number of false positives while identifying deadspots, the code analyzer uses a recursive algorithm shown in Figure 7. The last step shown in the algorithm (related to callers) is performed to identify indirect usages of a method of the input framework. SBC clusters the detected deadspot methods into their declaring classes.

5 Evaluation

We evaluated SpotWeb with eight widely used open source frameworks. In our evaluation, we investigate two research questions. First, what is the percentage

Subject	# Classes	# Methods	# Samples	# KLOC	URL
Log4j	207	1543	9768	2064	logging.apache.org/log4j
JUnit	56	531	8891	1558	www.junit.org
JGraphT	177	931	289	30	jgrapht.sourceforge.net
Grappa	44	561	2071	1978	www.graphviz.org
OpenJGraph	210	1365	1076	113	openjgraph.sourceforge.net
JUNG	461	3241	2390	353	jung.sourceforge.net
BCEL	357	3048	5225	1219	jakarta.apache.org/bcel
Javassit	249	2149	3226	631	www.csg.is.titech.ac.jp /chiba/javassist

Table 1. Subjects used for evaluating SpotWeb.

Subject	# Classes	Hotspots					Deadspots	
		# Classes	%	# Templ	# Hooks	# Depend	#Classes	%
Log4j	207	74	35.74	49	13	44	101	48.79
JUnit	56	23	41.07	11	5	11	20	35.71
JGraphT	177	41	23.16	20	9	0	125	70.62
Grappa	44	16	36.36	9	2	2	19	43.18
OpenJGraph	210	43	20.47	25	10	21	145	69.04
JUNG	461	228	49.45	144	29	225	144	31.23
BCEL	357	153	42.85	95	12	74	81	22.68
Javassit	249	77	30.92	60	8	27	126	50.60

Table 2. Evaluation results showing the detected hotspots and deadspots.

of hotspot and deadspot classes among the total number of classes? This research question helps to characterize the usages of a framework. Second, what is the effectiveness of our hotspot detection in terms of precision and recall? The subjects used in our evaluation and their characteristics such as the number of classes and methods are shown in Columns "Classes" and "Methods" of Table 1. Column "Samples" of Table 1 shows the number of code samples gathered and Column "KLOC" shows the total number of kilo lines of Java code analyzed by SpotWeb for identifying hotspots and deadspots. One of the major advantages of SpotWeb compared to other approaches is the large number of analyzed code samples that can help detect hotspots and deadspots effectively.

5.1 Statistics of Hotspots and Deadspots

We used SpotWeb to detect hotspots and deadspots of all subjects and the results of our evaluation are shown in Table 2. Column "Subject" shows the name of the input framework. The sub-columns "Classes" and "%" of Column "Hotspots" show the number of hotspot classes and their percentages among the total number of classes. Columns "Templ", "Hooks", and "Dependen" give the number of template hierarchies, hook hierarchies, and their dependencies, respectively. The sub-columns "Classes" and "%" of Column "Deadspots" show the number of deadspot classes and their percentages.

Our results show that the percentage of hotspots for all subjects ranges from 20% to 50%, whereas the percentage of deadspots ranges from 22% to 70%. Fig-

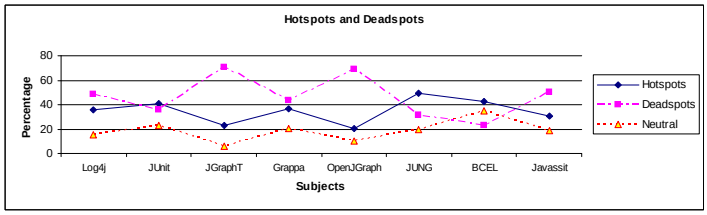

Fig. 8. Distribution of hotspot and deadspot percentages in all subject frameworks.

ure 8 presents the distribution of hotspot and deadspot percentages of all subjects. The distribution chart shows that OpenJGraph and JGraphT frameworks have the lowest percentage of hotspots and the highest percentage of deadspots. In the figure, we also show a new classification called "Neutral", which represents classes that do not belong to either the hotspot or deadspot category. The graph shows that the percentage of classes in the Neutral category is relatively low for all subjects except BCEL. This characteristic indicates that a class is either reused heavily or is never reused, and only in a few cases a class is occasionally reused.

5.2 Effectiveness of Hotspot Detection

We next analyze the effectiveness of hotspot detection using the evaluation results with Log4j and JUnit frameworks. The primary reason for selecting Log4j[2] and JUnit[3] for analysis is the availability of their documentation that can help validate the detected hotspots.

Log4j provides several features such as Appenders and Layouts, and for each such feature, Log4j provides several classes such as `ConsoleAppender` and `JDBCAppender` for the appender feature. Among those several classes provided for each feature, a few classes are much more often used than other classes. The features described in the documentation of Log4j are shown in Columns "Feature" and "Description" of Table 3. Column "Class" shows the commonly used classes for each feature. Each of these classes serves as starting points for using those features.

SpotWeb identified 74 classes as hotspots in the Log4j library, and these classes captured all 12 starting points described in the documentation resulting in a recall of 100%. In constrast, the precision is 16.21%. But the real hotspots could be beyond the starting points described in the documentation and our real precision could be much higher than the one calculated based on the documentation. Ideed, although the precision calculated based on the documentation can be increased by decreasing the HT threshold percentage used in our approach, we prefer to have a high recall. The primary reason is that SpotWeb sorts the detected hotspot classes based on their importance and ranks each class. Column "Rank" of Table 3 presents the rank among the total number of hotspots detected by SpotWeb. Column "Type" shows whether the detected hotspot is

[2] `http://logging.apache.org/log4j/docs/manual.html`
[3] `http://junit.sourceforge.net/doc/cookstour/cookstour.htm`

Feature	Description	Class	Rank	Type
Loggers	Log the messages of several levels	Category	1	TEMPLATE
		Logger	7	HOOK
		Level	12	HOOK
Appenders	Allows logging to multiple destinations	ConsoleAppender	8	TEMPLATE
		FileAppender	16	TEMPLATE
Layouts	Helps to format the logging request	PatternLayout	4	TEMPLATE
		SimpleLayout	11	TEMPLATE
Configurators	Helps to configure Log4j	BasicConfigurator	2	TEMPLATE
		PropertyConfigurator	3	TEMPLATE
		DOMConfigurator	5	TEMPLATE
Loaders	Helps to load resources	Loader	31	TEMPLATE
NDC	Nested diagnostic constant	NDC	10	TEMPLATE

Table 3. Hotspots described in Log4j documentation.

a `TEMPLATE` or a `HOOK`. Although SpotWeb has low precision, except the `Loader` class, all other 11 hotspot classes are ranked among the top 16 classes of the total 74 classes. Therefore, a user who plan to reuse APIs of the Log4j library can refer to the first 16 classes suggested by SpotWeb to identify where to start reusing the framework.

SpotWeb also captured the dependency information that is described in the documentation. For example, the appender classes of the Log4j library require layout classes. SpotWeb correctly identified a `TEMPLATE_TEMPLATE` dependency between appenders and layouts. The dependency describes that the user needs an instance of layouts such as `PatternLayout` or `SimpleLayout` to create an instance of appenders such as `ConsoleAppender` or `FileAppender`.

We used the cookbook provided with the JUnit framework to verify the detected hotspots. Hotspots detected in the JUnit framework are shown in Figure 1. SpotWeb identified all hotspot classes described in the cookbook resulting in a recall of 100%. The cookbook describes only six classes resulting in a precision of 26.08%. However, due to the sorting mechanism of SpotWeb, five of the hotspot classes described in the documentation are ranked among the top seven hotspot classes detected by SpotWeb. In addition, similar to the case in the documentation of the Log4j library, the real hotspots of JUnit could be beyond the ones described in the JUnit cookbook and our real precision could be higher than the one calculated based on the cookbook.

6 Threats to Validity

The threats to external validity primarily include the degree to which the subject programs and used CSE are representative of true practice. The current subjects range from small-scale applications such as Grappa to large-scale applications such as BCEL and JUNG. In SpotWeb, we used only one CSE, i.e., Google code search. We plan to reduce these threats by conducting more experiments on wider types of subjects and by using other CSEs in future work. The threats to internal validity are instrumentation effects that can bias our results. Faults in our SpotWeb prototype might cause such effects while detecting hotspots and

deadspots. To reduce these threats, we inspected some code samples gathered from the CSE to double check the metrics computed by SpotWeb for these code samples.

7 Conclusion

In this paper, we proposed an approach called SpotWeb that tries to assist software developers in reusing APIs of an existing framework by detecting hotspots and deadspots of the framework. Hotspots serve as starting points for reusing the framework, whereas deadspots serve as caveats. SpotWeb tries to address major problems faced by earlier related approaches by not requiring any additional efforts from the developers and by collecting relevant code samples through a code search engine. We evaluated our approach through eight popular open source frameworks and showed that SpotWeb can give a high recall while detecting hotspots and the detected hotspots of Log4j and JUnit frameworks are consistent with the starting points described in the documentation for these frameworks.

References

1. G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of Java software. In *Proc. OOPSLA*, pages 397–412, 2006.
2. N. Flores, D. Soares, H. Ferreira, and M. Rodrigues. HotSpotter: a JavaML-based approach to discover Framework's Hotspots. In *Proc. XATA*, 2005.
3. G. Froehlich, J. Hoover, L. Liu, and P. Sorenson. Hooking into object-oriented application frameworks. In *Proc. ICSE*, pages 491–501, 1997.
4. Google Code Search Engine, 2006. `http://www.google.com/codesearch`.
5. R. Johnson. Documenting frameworks using patterns. In *Proc. OOPSLA*, pages 63–76, 1992.
6. Junit, 2001. `http://www.junit.org`.
7. The Koders source code search engine, 2005. `http://www.koders.com`.
8. The Krugle code search for developers, 2006. `http://www.krugle.com`.
9. T. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. In *IEEE Software*, pages 35–39, 2003.
10. R. Martin. The Open Closed Principle. *j-C-PLUS-PLUS-REPORT*, 8(1):37–43, 1996.
11. M. Mendonca, P. Alencar, T. Oliveira, and D. Cowan. Assisting aspect-oriented framework instantiation: towards modeling, transformation and tool support. In *Proc. OOPSLA*, pages 94–95, 2005.
12. W. Pree. Meta Patterns - A Means For Capturing the Essentials of Reusable Object-Oriented Design. In *Proc. ECOOP*, pages 150–162, 1994.
13. S. Thummalapenta and T. Xie. PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proc. ASE*, 2007.
14. T. Xie and J. Pei. MAPO: Mining API usages from open source repositories. In *Proc. MSR*, pages 54–57, 2006.