SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web

Suresh Thummalapenta
Department of Computer Science
North Carolina State University
Raleigh, USA
sthumma@ncsu.edu

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, USA
xie@csc.ncsu.edu

***Abstract*—Software developers often face challenges in reusing open source frameworks due to several factors such as the framework complexity and lack of proper documentation. In this paper, we propose a code-search-engine-based approach that detects *hotspots* in a given framework by mining code examples gathered from open source repositories available on the web; these hotspots are API classes and methods that are frequently reused. Hotspots can serve as starting points for developers in understanding and reusing the given framework. Our approach also detects *coldspots*, which are API classes and methods that are rarely used. Coldspots serve as caveats for developers as there can be difficulties in finding relevant code examples and are generally less exercised compared to hotspots. We developed a tool, called SpotWeb, for frameworks or libraries written in Java and used our tool to detect hotspots and coldspots of eight widely used open source frameworks. We show the utility of our detected hotspots by comparing these hotspots with the API classes reused by a real application and compare our results with the results of a previous related approach.**

I. 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 often offer complex API methods that pose challenges to developers 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the documentation is often missing for many existing frameworks and even if such documentation exists, it is often outdated [1].

In general, frameworks expose certain areas (API classes and methods) of flexibility that are intended for reuse by their users. Software developers who reuse classes and methods 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 [2], hotspots depict a framework's flexibility and proneness to reuse. The foundations of hotspots are built upon the Open-Closed principle by Martin [3]. The Open-Closed principle encompasses two main definitions: the "open" and the "closed" parts. The "open" parts (referred as *hooks*) represent areas that are flexible and variant, whereas the "closed" parts (referred as *templates*) represent areas that are immutable in the given framework. A *hotspot* is defined as a combination of templates and hooks.

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 classes and find out the classes 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 application code that reuses framework 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 effort (e.g., performance improvement or bug fixing) 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 *coldspots*. The concept of coldspots is introduced by our approach and these coldspots can serve as caveats to users of the given framework. As coldspots represent the rarely used classes and methods, there can be difficulties in identifying relevant code examples that can help users in reusing those classes and methods. Moreover, coldspots are generally less tested compared to hotspots with regards to the "testing" conducted by API client code as test code.

Detecting hotspots and coldspots of an input framework requires domain knowledge of how the API classes and methods of the input framework are reused by applications, referred as client applications. Various open source projects that reuse classes of a given input framework are available on the web and these open source projects can serve as a basis for gathering the information of how classes of the input framework are reused, and hence can help in detecting hotspots and coldspots. Therefore, our approach, called SpotWeb, leverages a code search engine (CSE) to gather relevant code examples of classes of the input framework from these open source

projects. Given a query, a CSE can extract code examples with usages of the query from open source projects available on the web. Our approach analyzes gathered code examples statically and detects hotspots and coldspots of the given framework. Our approach tackles the problems related to the quality of code examples gathered from a CSE by capturing the most common usages of classes through mining.

The paper[1] makes the following main contributions:

- An approach for detecting hotspots of a given framework by analyzing relevant code examples gathered from a CSE.
- An approach for detecting coldspots of a given framework.
- A tool (as an Eclipse plugin) implemented for the proposed approach and several evaluations to assess the effectiveness of the tool. In our evaluation, SpotWeb detects hotspots and coldspots of eight widely used open source frameworks by analyzing a total of 7.9 million lines of code. We show the utility of detected hotspots by comparing detected hotspots of a framework with a real application reusing that framework. We also compare our results with the results of a previous related approach by Viljamaa [5].

The rest of the paper is organized as follows. Section II explains our approach through an illustrative example. Section III describes key aspects of the approach. Section IV presents implementation details. Section V discusses evaluation results. Section VI presents threats to validity. Section VII presents related work. Finally, Section VIII concludes.

II. Example

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

SpotWeb accepts an input framework, say JUnit, and extracts *FrameworkInfo* from the framework. The *FrameworkInfo* 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 also captures the constants defined by the input framework. SpotWeb constructs different queries for each class or interface and interacts with a CSE such as Google code search [7] to gather relevant code examples from existing open source projects that reuse the classes of the input framework. For example, SpotWeb constructs a query such as "`lang:java junit.framework.TestSuite`" for gathering relevant code examples of the `TestSuite` class. These gathered code examples are referred as a *LocalRepository* for the input framework. SpotWeb analyzes gathered code examples 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 5 for detecting hotspots from the computed *UsageMetrics*.

Initially, SpotWeb ranks methods in a non-ascending order based on their *UsageMetrics* and uses a threshold percentage HT to detect hotspot methods: the methods in the top HT percentage with a non-zero *UsageMetrics* are detected as hotspot methods. The detected hotspot methods are then grouped 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 Step 4 of the algorithm shown in Figure 5. The hotspot classes of each category are further grouped 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 groups 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 1 and the `TestCase` class has Rank 2, and then the grouped hierarchy of the `Assert` and `TestCase` classes is assigned with Rank 1. Hierarchies with smaller ranks have higher preference or importance to the hierarchies with larger ranks.

Figure 1 shows the hotspot hierarchies detected for the JUnit framework. The figure also shows ranks assigned to each hierarchy. Each hierarchy includes one or more hotspot classes and is shown as pairs of a class and its methods. For example, Hierarchy 1 (hierarchy with Rank 1) has classes `Assert`, `TestCase`, `TestSetup`, and `TestDecorator`. 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 a hierarchy are sorted based on their computed *UsageMetrics*. Sorting 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 `<init>(Class)` constructor is often used compared to the other two constructors. Due to space limit, we show all assertion methods such as `assertEquals` and `assertTrue` of the class `Assert` of Hierarchy 1 as `assertXXX`.

The figure also displays dependencies among hotspot hierarchies (shown as arrows between hierarchies). For example, Hierarchy 5 has a `TEMPLATE_HOOK` dependency with Hierarchy 1. This dependency indicates that to reuse methods such

[1] An earlier version of this work is described in a position paper presented at MSR 2008 [4].

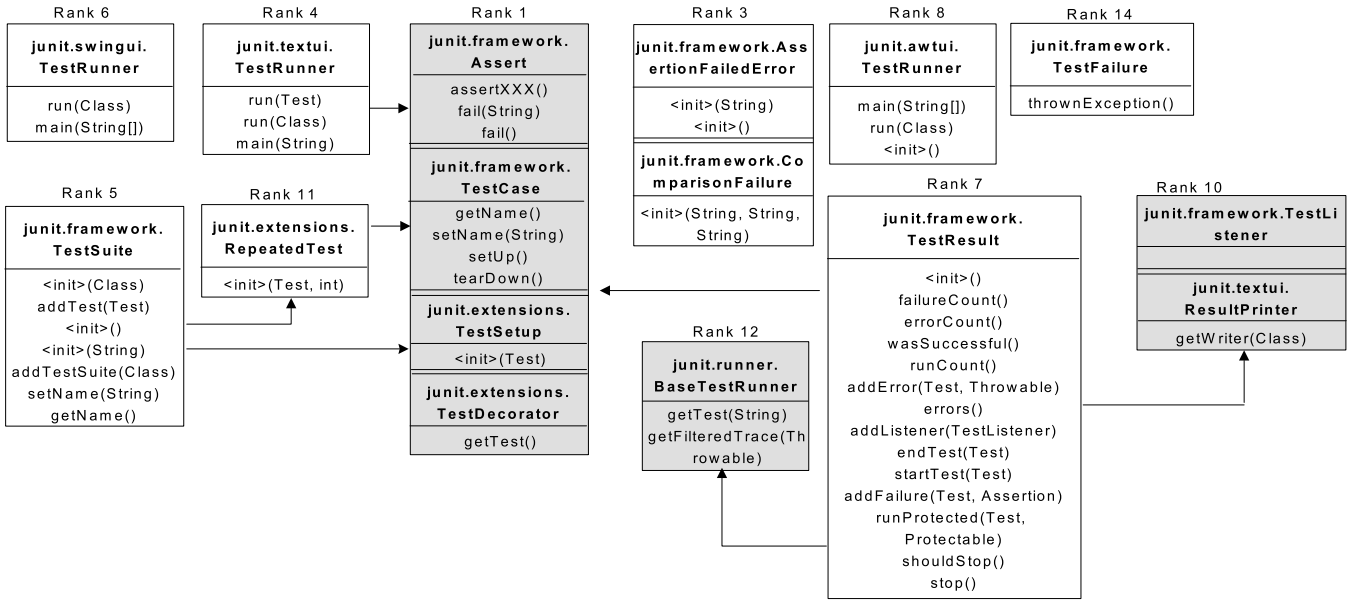

Fig. 1. Hotspot hierarchies identified for the JUnit framework

```
01:public class SRDAOTestCase
02:      extends TestCase {
03:  private SRDAO dao = null;...
04:  public SRDAOTestCase() {
05:     super(); ...
06:  }
07:  protected void setUp() throws Exception {
08:     ...
09:     dao = (SRDAO)context.getBean("SRDAO");
10:     ...
11:  }
12:  public void tearDown() throws Exception {
13:     dao = null;
14:  }
15:  public void testF() { ... }
16:  public void testB() { ... }
17:  ...
18:}
```

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

```
01:public class MyTestSuite {
02:  ...
03:  public static Test suite() {
04:     TestSuite suite = new TestSuite("axis");
05:     suite.addTest(new SRDAOTestCase());
06:     return suite;
07:  }
08:  ...
09:}
```

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

as `addTest` of the class `TestSuite` in Hierarchy 5, the user has to define a new behavior for the classes in Hierarchy 1 because the first argument of `addTest` requires instances of classes such as `TestCase` of Hierarchy 1.

We next describe how the hotspots detected by SpotWeb can be used by the framework users to reuse classes 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 part of Hierarchy 1. The code example exhibits several aspects that need 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. In addition, the code example shows that names of the test methods in the extended class of the `TestCase` class should start with the prefix `test`. SpotWeb also recommends a code example for the `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 class or method is identified as a coldspot if that class or method is neither used directly nor used indirectly by gathered code examples. The complete algorithm used for detecting coldspots is shown in Figure 7. SpotWeb identified 20 classes such as `Swapper`, `TestRunListener`, and `ExceptionTestCase` as coldspots in the JUnit framework. However, coldspots are only suggestions for users unfamiliar to that framework and SpotWeb does not intend to recommend users not to reuse those coldspot classes. Sometimes, coldspots can also be helpful to the framework developers in distributing their maintenance effort, because the framework developers can give a low preference to the coldspot classes.

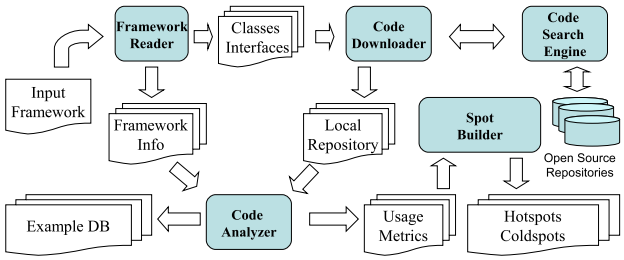

Fig. 4. Overview of the SpotWeb approach

III. Approach

Our approach consists of five major components: the framework reader, CSE, code downloader, code analyzer, and spot builder. Figure 4 shows an overview of all components and flows among different components. We use JUnit as an illustrative example for explaining our approach.

A. Framework Reader

The framework reader component takes a framework, say JUnit, as input and extracts the *FrameworkInfo* information. The *FrameworkInfo* includes all classes, all interfaces, public or protected methods of each class and interface.

B. Code Downloader

The code downloader interacts with a CSE to download relevant code examples. For example, the code downloader constructs a query such as "`lang:java junit.framework.TestSuite`" for gathering relevant code examples of the `TestSuite` class. The downloaded code examples, referred as *LocalRepository*, are given as input to the code analyzer. The code examples stored in the *LocalRepository* are often partial as CSE gathers individual source files, instead of entire projects. In our approach, we used Google code search [7] for gathering relevant code code examples. There are two main reasons for using Google code search as an underlying search engine in our approach: Google code search provides open APIs to interact through client code and is well-maintained.

C. Code Analyzer

The code analyzer analyzes code examples stored in the *LocalRepository* statically and computes *UsageMetrics* for all classes, methods, exceptions, and constants of the input framework. As these code examples are partial, the code analyzer uses several type heuristics for resolving object types. These type heuristics are described in our previous approach, called PARSEWeb [8]. The *UsageMetrics* capture several ways of how often a class or an interface or a method of the input framework is used by gathered code examples.

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 *UsageMetrics* for an exception class include the number of times that the exception class is used in the `catch` blocks or the `throw` statements.

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 examples. The *Overrides* metric gives the number of times that the method is overridden by the code examples to define a new behavior. The *Implements* metric, specific for interfaces, gives the number of times that the method is implemented. The code analyzer considers `static` methods as regular methods. 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 (UM_i) for a method is the sum of all the three preceding metrics.

The code analyzer identifies constants defined by the input framework through the Java keywords such as `final` and `static`. The *UsageMetrics* for such a constant include the number of times that the constant is referred by code examples gathered from CSE.

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 recommending relevant code examples for a class or a method requested by the users. The relevant code examples can further assist the users in making effective reuse of API classes and methods of the input framework.

D. Spot Builder

The spot builder component (SBC) analyzes gathered code examples and detects hotspots and coldspots.

1) Identification of hotspots: The spot builder component (SBC) uses computed *UsageMetrics* for detecting hotspots. The algorithm used by SBC for detecting hotspots is shown in Figure 5. We next describe the algorithm through an illustrative example shown in Figure 6. The figure shows four classes C_1, C_2, C_3, and `ExceptClass`, and their declarations. The class C_3 is an `abstract` class. The `ExceptClass` is an `Exception` class that can appear in exception-handling constructs such as `catch` blocks. The figure also shows computed usage metrics for each class, and its methods and constant variables. 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). The method m_{2_1} is invoked for 6 times and is overridden for 2 times. Similarly, the constant `constC1` is accessed 6 times and the exception class `ExceptClass` is detected in `catch` blocks for 9 times among gathered code examples.

Initially, SBC sorts UM values of all methods, constants, and exception classes. SBC uses a threshold percentage (referred as HT) and selects the top HT methods, whose usage metric is non-zero, as hotspot methods. For example, for a

Input: UsageMetrics of classes and methods, HT percentage
Output: Hotspot hierarchies and their dependencies
1:*SortedMET* = Sort methods based on their usage metric values;
2:foreach MET_i in $SortedMET$ {
    if ($UM_i \neq 0$)
        if (Position of $MET_i \leq$ (HT * Size of $SortedMET$))
            Set MET_i type as $HOTSPOT$;
    }
3:{C_1, ... C_n} = Group $HOTSPOT$ MET_i based on their declaring classes;
//Assign ranks to each C_i and classify into templates and hooks
4:foreach C_i in {C_1, ... C_n} {
    Rank of C_i = Minimum rank among all MET_i of the C_i;
    if C_i is an *Interface* or *Abstract class* or ($EX_i > IN_i$)
        Set type of C_i to $HOOK$;
    otherwise
        Set type of C_i to $TEMPLATE$;
    }
5:Group C_i of the same type into hierarchies based on inheritance;
6:Associate hook hierarchies to template hierarchies;
7:Define dependencies between template hierarchies;
8:Output hook and template hierarchies as hotspot hierarchies;

Fig. 5. Algorithm used for detecting hotspots through computed *UsageMetrics*

```
class C1 {
    /*IN = 10,EX=0,IM = 0*/

 C1 () { … }
    /*IN = 10,OV=0,IM = 0*/

 m1_1 (C3 arg1) { … }
    /*IN = 8,OV = 0,IM = 0*/

 m1_2 () { … }
    /* IN = 3, OV=0,IM=0*/

 final static constC1;
    /*  UM = 6 */
}
```

```
class C2 {
    /*IN = 6,EX=2,IM = 0*/

 C2 (C1 arg1) { … }
    /*IN = 6,OV=0,IM = 0*/

 m2_1 (C3 arg1) { … }
    /*IN = 6,OV = 2,IM = 0*/

 m2_2 () { … }
    /*IN = 1,OV = 2,IM = 0*/

 final static constC2;
    /* UM = 1 */
}
```

```
abstract class C3 {
    /*IN = 0,EX=12,IM = 0*/

 abstract m3_1 ();
    /*IN = 0,OV=12,IM = 0*/

 abstract m3_2 ();
    /*IN = 0,OV=12,IM = 0*/

 abstract m3_3 ();
    /*IN = 0,OV=12,IM = 0*/
}
```

```
class ExceptClass
  extends Exception
{
    /* UM = 9 */
}
```

Fig. 6. Example classes of a sample framework

HT of 45%, SBC identifies the methods such as m_{3_1}, m_{3_2}, m_{3_3}, and c_1 as hotspot methods. SBC groups the hotspot methods based on their declaring classes. The resulting classes are sorted based on the minimum rank among included hotspot methods in each class. In the current example, the grouping

Input: A method M_i of a class C_j
Output: Is the method a coldspot or not?
1:Return false if the method is reused atleast once;
2:if C_j is an interface
    Return true if all implemented methods of M_i are coldspots;
    Otherwise return false;
3:if M_i is abstract
    Return true if all overridden methods of M_i are coldspots;
    Otherwise return false;
4:Return true if all callers of M_i are coldspots
    Otherwise return false;

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

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 grouping, SBC uses 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 Step 4 of the algorithm shown in Figure 5. 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 groups the classes of the same category based on their inheritance relationship. For example, if C_1 has a parent class P_1 and both classes are classified as `TEMPLATE` classes, SBC groups C_1 and P_1 into the same hierarchy.

SBC identifies dependencies among the detected hotspot hierarchies based on arguments passed to methods of those classes. 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. 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 types include some classes in 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 classes, and use the instances of those classes as arguments. For example, SBC identifies that the 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, SBC identifies `TEMPLATE_TEMPLATE` hierarchies when one template hierarchy is dependent on another template hierarchy. For example, the class C_2 has a `TEMPLATE_TEMPLATE` dependency with the class C_1.

2) Identification of coldspots: SBC identifies classes and methods (of the input framework) that are rarely or never used by gathered code examples as coldspots. However, detecting coldspots 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 gathered code examples refer to the concrete implementation provided by

some of the abstract classes's subclasses. In this case, this abstract method is not a coldspot as the method is indirectly referenced through the subclasses. Therefore, to reduce the number of false positives while identifying coldspots, the code analyzer uses a recursive algorithm shown in Figure 7. Step 4 of the algorithm (related to callers) is performed to identify indirect usages of a method of the input framework. SBC groups detected coldspot methods into their declaring classes.

IV. Implementation

We developed SpotWeb as an Eclipse plugin. SpotWeb accepts an input framework in the form of an Eclipse project. SpotWeb requires the input framework to be compilable, i.e., all dependent jars have to be specified in the classpath of the input framework. In the SpotWeb implementation, we used the HT percentage of 15%, which is derived based on our initial empirical experience.

V. Evaluation

We evaluated SpotWeb with eight widely used open source frameworks, which differ in size and purpose. In our evaluation, we investigate the following research questions.

- What is the percentage of hotspot and coldspot classes and methods among the total number of classes and methods in each framework, respectively? This research question helps to characterize the usages of a framework and effort in learning to reuse the framework.
- Is the subset of classes and methods detected as hotspots indeed useful in helping effective framework reuse? We address the preceding question by showing that detected hotspots include classes and methods of a framework reused by a real application. This evaluation helps to show that our approach can help reduce the effort of users by suggesting a subset of classes and methods as hotspots.
- What is the effectiveness of our hotspot detection in terms of precision and recall? We address the preceding question through two evaluations. First, we compare the detected hotspot classes with the classes described in the documentation associated with the frameworks. Second, we compare the detected hotspot classes with the hotspots detected by a previous related approach by Viljamaa [5].

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 I. Column "Samples" of Table I shows the number of code examples gathered from Google code search [7] and Column "KLOC" shows the total number of kilo lines of Java code analyzed by SpotWeb for identifying hotspots and coldspots. One of the major advantages of SpotWeb compared to other approaches is the large number of analyzed code examples that can help detect hotspots and coldspots effectively.

A. Statistics of Hotspots and Coldspots

We next address the first question on the percentage of classes and methods classified as hotspots or coldspots. These statistics help identify the possibilities of reuse in a framework and the amount of effort required by a new user in getting familiar with the given framework. Table II shows the statistics of hotspots in all frameworks. Column "Subject" shows the name of the input framework. Columns "Hotspot classes" and "Coldspot classes" present the number of classes classified as hotspots and coldspots, respectively. Column "Hotspot methods" shows the number of methods identified as hotspot methods. Sub-columns "Classes" and "%" of Column "Hotspot classes" show the number of hotspot classes and their percentages among the total number of classes. Columns "Templ", "Hooks", and "Depend" give the number of template hierarchies, hook hierarchies, and their dependencies, respectively. Sub-columns "Classes" and "%" of Column "Coldspots" show the number of coldspot classes and their percentages.

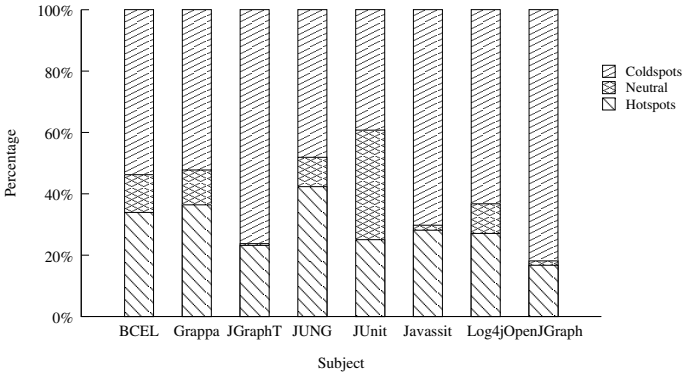

Fig. 8. Distribution of hotspot and coldspot percentages in all subject frameworks.

Our results show that the percentage of hotspots for all subjects ranges from 16% to 42%, whereas the percentage of coldspots ranges from 39% to 82%. These statistics help characterize the effort in reusing a given framework. For example, the required effort for learning to reuse the JUnit framework can be low compared to the required effort for learning to reuse the JUNG library because the number of hotspots of the JUNG library is greater than the number of hotspots of the JUnit framework; these hotspots can often be browsed or investigated by the users to learn how to reuse the framework. Figure 8 presents the distribution of hotspot and coldspot percentages of all subjects. The distribution chart shows that OpenJGraph and JGraphT frameworks have the lowest percentage of hotspots and the highest percentage of coldspots. This scenario can provide a hint that only a few classes of these frameworks are often reused. In the figure, we also show a new classification called "Neutral", which represents classes that do not belong to either the hotspot or coldspot category. The graph shows that the percentage of classes in the Neutral category is relatively low for all subjects except JUnit. 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.

We next describe a few example hotspot classes detected for the four graph libraries JGraphT, Grappa,

TABLE I
SUBJECTS USED FOR EVALUATING SPOTWEB.

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
TOTAL	1761	13369	32936	7946	

TABLE II
EVALUATION RESULTS SHOWING THE DETECTED HOTSPOTS AND COLDSPOTS.

Subject	# Classes	Hotspot classes					Coldspot classes		Hotspot methods		
		# Classes	%	# Templ	# Hooks	# Depend	#Classes	%	# Total Methods	# Hotspot Methods	%
Log4j	207	56	27.05	35	11	22	131	63.28	1543	299	19.38
JUnit	56	14	25	8	3	7	22	39.28	531	77	14.50
JGraphT	177	41	23.16	20	8	0	135	76.27	931	102	10.96
Grappa	44	16	36.36	11	2	3	23	52.27	561	50	8.91
OpenJGraph	210	35	16.67	21	6	7	172	81.90	1365	76	5.57
JUNG	461	195	42.29	128	17	109	222	48.15	3241	569	17.55
BCEL	357	121	33.89	74	9	59	192	53.78	3048	580	19.03
Javassit	249	70	28.11	57	4	14	175	70.28	2149	371	17.26

OpenJGraph, and Grappa. Each graph library provides several different types of graphs. For example, the JUNG library provides graphs such as `DirectedGraph`, `ArchetypeGraph`, and `HyperGraph`. SpotWeb identified the graph type that is commonly used among all different types provided by each library. For example, SpotWeb identified that graphs `DefaultListenableGraph`, `DirectedSparseGraph`, `DirectedGraphImpl`, and `Graph` are the commonly used graph types in JGraphT, JUNG, OpenJGraph, and Grappa graph libraries, respectively.

B. Utilities of Hotspots

We next try to address the second question on whether the subset of classes and methods detected as hotspots is indeed useful in helping effective framework reuse. We use DNSJava[2], a popular framework that provides implementation of DNS in Java, as an input framework. We choose DNSJava for two primary reasons: DNSJava is used as a subject in several previous approaches and the DNSJava webpage provides example applications that can be used to validate the detected hotspots. We classify all DNSJava-reusing applications available on the web through Google code search (except the James[3] application) as training applications for SpotWeb to detect hotspots of DNSJava. We validate the detected hotspots using James as a test application. We selected James as the test application, because James is one of the example applications described in the webpage of DNSJava.

To show the utility of the detected hotspots, we identify the DNSJava classes and methods that are reused by James and compute the percentage of those classes detected by SpotWeb. DNSJava includes 151 classes and 1224 methods. The James application reused 17 classes and 18 methods of DNSJava. James also used 1 exception class and 7 constants declared by DNSJava. SpotWeb identified 47 classes, 190 methods, 3 exceptions, and 50 constants as hotspots in DNSJava. Table III shows our evaluation results in terms of precision and recall.

TABLE III
HOTSPOTS OF DNSJAVA REUSED BY JAMES.

	James	SpotWeb	Precision	Recall
Classes	17	16	32	94.11
Methods	18	16	8.42	88.88
Exceptions	1	1	33.33	100
Constants	7	7	14	100

Column "James" shows the number of classes, methods, exceptions, and constants used by James. Column "SpotWeb" shows the number of the ones (used by James) that are among detected hotspots by SpotWeb. The hotspots detected by SpotWeb include 16 classes and 16 methods reused by James. Moreover, SpotWeb also correctly detected all 7 constants and 1 exception class referred by James. This evaluation shows that the subset of classes and methods detected as hotspots are indeed useful and can help reduce the effort of a programmer unfamiliar to the framework.

SpotWeb could not detect one hotspot class, called `Resolver`. The primary reason is that `Resolver` is an interface and gathered code examples do not include any usages of the `Resolver` interface. The results show that SpotWeb has a high recall and low precision with respect to the James application. In our approach, we incline toward a high recall

[2]http://www.dnsjava.org/
[3]http://james.apache.org/

because with a high recall, framework users would miss only few hotspot classes. We discuss more on how we alleviate low precision in the subsequent evaluation.

C. Effectiveness of Hotspot Detection

We next address the third question regarding the effectiveness of our hotspot detection with respect to precision and recall through two evaluations. First, we compare the detected hotspots of JUnit and Log4j frameworks with available documentations. Second, we compare SpotWeb results with the results of a previous approach by Vijamaa [5].

1) Comparison with documentation: We next analyze the effectiveness of hotspot detection through the evaluation results with Log4j and JUnit frameworks. The primary reason for selecting Log4j[4] and JUnit[5] 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. For example, Log4j provides classes `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 IV. Column "Class" shows the commonly used classes for each feature. Each of these classes serves as starting points for using those features.

SpotWeb identified 56 classes as hotspots in Log4j for the HT percentage of 15%, and these classes captured all 12 starting points described in the documentation resulting in a recall of 100%. In constrast, the precision is 21.42%. Column "Rank" of Table IV presents the rank of each documented class among the total number of hotspots detected by SpotWeb. Column "Type" shows whether the detected hotspot is a `TEMPLATE` or a `HOOK`. The table also shows that except the `Loader` class, all other 11 hotspot classes are ranked among the top 16 classes of the total 56 classes. Therefore, a user who plans to reuse classes of Log4j can refer to the first 16 classes suggested by SpotWeb to identify where to start reusing the framework.

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 5 out of 6 hotspot classes described in the cookbook resulting in a recall of 83.33% and precision 35.71%.

We next describe our empirical analysis with several HT values and describe why we use HT of 15% to identify hotspots. Figure 9 shows the results with three subjects and several threshold values ranging from 5% to 25%. With the increase in the threshold value, the precision decreases and the recall increases. Although decrease in the precision is common with increase in the threshold value, the figure shows that the precision decreases rapidly with increase in the threshold value from 5% to 10%. This observation indicates that many commonly reused API methods (even among the top 10% of ranked API methods detected by SpotWeb based on usage) are not included in the documentation. This phenomenon shows that the real hotspots of these frameworks are beyond those hotspots calculated based on the documentation because the documentation of existing frameworks is often outdated or incomplete. This evaluation shows the necessity of an approach such as SpotWeb that can automatically infer hotspots from the applications that are already reusing the API classes and methods of the input framework.

In our implementation, we used HT as 15% as this threshold value has high recall with reasonable precision. The primary reason for inclining toward a high recall with low precision is that the framework users would miss few hotspot classes. To alleviate the low precision, our ranking mechanism helps give higher priority to those hotspot classes that are described in the documentation compared to other classes as shown in Table IV.

2) Comparison with Viljamaa's approach: We next compare the results of SpotWeb with the results of a previous related approach by Viljamaa [5] for the JUnit framework. Their approach also recovers the hotspots of a framework by using the source code of the framework and a set of available example applications. Their approach uses concept analysis [9] for recovering hotspots of the framework. As both approaches target a similar problem, we compared the results of SpotWeb for the JUnit framework with the results of their approach. Viljamaa's approach detected a total of 7 classes as hotspot classes for JUnit framework, whereas SpotWeb detected 14 classes as hotspot classes. Section V-C1 presents the precision and recall values for SpotWeb with respect to the JUnit documentation.

SpotWeb detected 5 classes among the 7 classes detected by Viljamaa's approach. The 2 missing classes are `ExceptionTestCase` and `ActiveTestSuite`. However, both these classes are not described in the available documentation of JUnit. SpotWeb is not able to detect these classes because of low support among gathered code examples. SpotWeb detected 4 new classes and several new methods that are not detected by Viljamaa's approach and are described in the documentation. For example, classes of JUnit such as `Assert`, `TestResult`, and `TestFailure` (shown in Figure 1) are often used and detected only by SpotWeb. Similarly, for the `Testcase` class, the `tearDown` method is also often used along with the `setUp` method. Viljamaa's approach detected only the `setUp` method, whereas SpotWeb detected both `setUp` and `tearDown` methods as hotspot methods. These results show that SpotWeb can perform better than Viljamaa's approach. Furthermore, Viljamaa's approach requires the users to have some initial knowledge of the structure and hotspots of the input framework that is being analyzed. In contrast, our approach does not require the users to have any knowledge regarding the input framework.

[4]http://logging.apache.org/log4j/docs/manual.html

[5]http://junit.sourceforge.net/doc/cookstour/cookstour.htm

TABLE IV
HOTSPOTS DESCRIBED IN THE LOG4J DOCUMENTATION.

Feature	Description	Class	Rank	Type
Loggers	Log the messages of several levels	Category	1	TEMPLATE
		Logger	9	HOOK
		Level	4	HOOK
Appenders	Allows logging to multiple destinations	ConsoleAppender	11	TEMPLATE
		FileAppender	16	TEMPLATE
Layouts	Helps to format the logging request	PatternLayout	5	TEMPLATE
		SimpleLayout	13	TEMPLATE
Configurators	Helps to configure Log4j	BasicConfigurator	2	TEMPLATE
		PropertyConfigurator	3	TEMPLATE
		DOMConfigurator	7	TEMPLATE
Loaders	Helps to load resources	Loader	27	TEMPLATE
NDC	Nested diagnostic constant	NDC	12	TEMPLATE

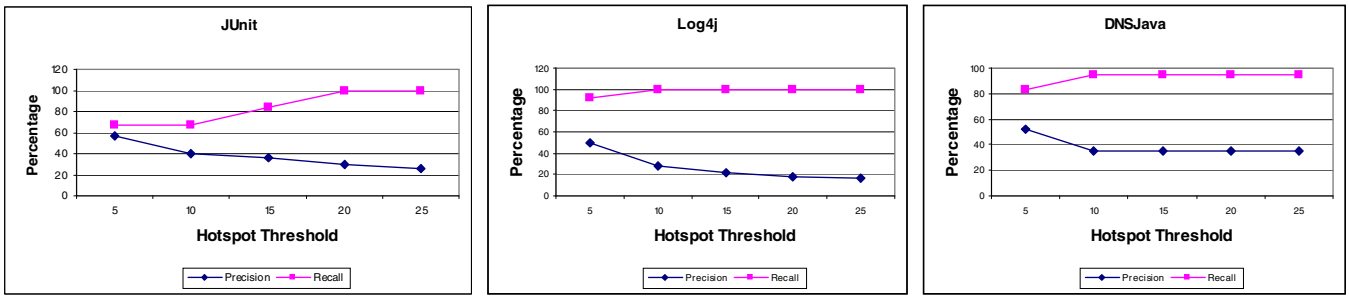

Fig. 9. Precision and Recall for five HT values with JUnit, Log4j, and DNSJava

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

VII. RELATED WORK

We used CSEs for gathering related code examples in our previous approaches called MAPO [10] and PARSEWeb [8]. With MAPO and PARSEWeb, SpotWeb shares *only* the code downloader component that is used for gathering related code examples. As each approach targets a different problem, each approach has different techniques in analyzing gathered code examples to address the challenges of that problem. MAPO and PARSEWeb focus on mining API usage patterns to assist programmers to write effective API client code. Given an API method, MAPO identifies the frequent usage patterns of that API method from gathered code examples. PARSEWeb analyzes each gathered code example and builds a CFG, which includes only method-invocation nodes, and captures sequences that serve as solutions for the queries of the form "Source object type → Destination object type". The SpotWeb approach is significantly different from these approaches as SpotWeb targets at capturing *UsageMetrics* of classes and methods of a given input framework and uses these metrics to detect hotspots and coldspots.

An approach by Viljamaa [5] also recovers the hotspots of a framework by using the source code of the input framework and a set of available example applications. Their approach uses concept analysis [9] for uncovering hotspots of the framework. One major problem with their approach is that applying concept analysis to the entire input source code can result in a huge pattern that is not useful in practice. To address the preceding problem, their approach suggests to select only those program elements that are relevant to the hotspot h at hand. Therefore, their approach requires the users to have some initial knowledge of the structure and hotspots of the framework under analysis. In contrast, our approach uses simple statistical analysis and can handle an entire input framework. Furthermore, SpotWeb does not require the users to have any knowledge of the input framework. Finally, our approach performs better than their approach as shown in our evaluation.

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 framework developers to manually specify the framework design in a specific process language, called Reuse Definition Language, proposed by their approach.

Holmes and Walker [12] proposed an approach that quantitatively determines how existing APIs are used. Their approach gathers a few applications that already reuse those existing APIs and computes metrics to detect how existing APIs are used by those applications. SpotWeb differs from their approach in three main aspects. First, their approach expects the framework users to have knowledge of the APIs of the framework. Therefore, their approach is mainly useful to users who are already familiar with those framework APIs. Second, their approach presents only the number of times that the APIs are reused. Third, their approach computes metrics from a limited data scope. In contrast, SpotWeb does not require the users to have the knowledge of APIs of the input framework and presents information in a more comprehensive form through templates and hooks.

Baxter et al. [13] 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 framework developers who can evaluate the structural features of their own programming practice and optimize their performance. In contrast, SpotWeb is useful for framework users in effectively reusing the APIs of a framework.

VIII. Conclusion

In this paper, we proposed an approach called SpotWeb that assists software developers in reusing API classes and methods of an existing framework by detecting hotspots and coldspots of the framework. SpotWeb addresses major problems faced by previous related approaches by not requiring any additional effort from the developers and by collecting relevant code examples through a code search engine. In our evaluation, we found that often classes of existing frameworks are either reused heavily or never reused. We also showed that the detected hotspots are indeed useful in helping effective framework reuse and showed that SpotWeb performs better than an earlier related approach. In future work, we plan to explore other applications of hotspots such as prioritization of bug reports based on hotspots.

Acknowledgments

This work is supported in part by NSF grants CNS-0720641 and CCF-0725190, and Army Research Office grant W911NF-07-1-0431.

References

[1] T. C. Lethbridge, J. Singer, and A. Forward, "How Software Engineers use Documentation: The State of the Practice," in *IEEE Software*, vol. 20, no. 6. Los Alamitos, CA, USA: IEEE Computer Society Press, 2003, pp. 35–39.

[2] W. Pree, "Meta Patterns - A Means For Capturing the Essentials of Reusable Object-Oriented Design," in *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP)*. London, UK: Springer-Verlag, 1994, pp. 150–162.

[3] R. C. Martin, "The Open Closed Principle," *j-C-PLUS-PLUS-REPORT*, vol. 8, no. 1, pp. 37–43, 1996.

[4] S. Thummalapenta and T. Xie, "SpotWeb: Detecting Framework Hotspots via Mining Open Source Repositories on the Web," in *Proceedings of the 2008 International Workshop on Mining Software Repositories (MSR)*. New York, NY, USA: ACM, 2008, pp. 109–112.

[5] J. Viljamaa, "Reverse Engineering Framework Reuse Interfaces," in *Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. New York, NY, USA: ACM, 2003, pp. 217–226.

[6] "JUnit," 2001, http://www.junit.org.

[7] "Google Code Search," 2006, http://www.google.com/codesearch.

[8] S. Thummalapenta and T. Xie, "PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. New York, NY, USA: ACM, 2007, pp. 204–213.

[9] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997, translator-C. Franzke.

[10] T. Xie and J. Pei, "MAPO: Mining API Usages from Open Source Repositories," in *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR)*. New York, NY, USA: ACM, 2006, pp. 54–57.

[11] M. Mendonca, P. Alencar, T. Oliveira, and D. Cowan, "Assisting Aspect-Oriented Framework Instantiation: Towards Modeling, Transformation and Tool Support," in *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. New York, NY, USA: ACM, 2005, pp. 94–95.

[12] R. Holmes and R. J. Walker, "Informing Eclipse API Production and Consumption," in *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange (ETX)*, 2007, pp. 70–74.

[13] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero, "Understanding the Shape of Java Software," *SIGPLAN Not.*, vol. 41, no. 10, pp. 397–412, 2006.