SpotWeb: Detecting Framework Hotspots via Mining Open Source Repositories 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

The essentials of modern software development (such as low cost and high efficiency) demand software developers to make intensive reuse of 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 by mining code examples gathered from open source repositories available on the web; these hotspots are the APIs that are frequently reused. Hotspots can serve as starting points for developers in understanding and reusing the given framework. We developed a tool, called SpotWeb, for frameworks or libraries written in Java and conducted two case studies with two open source frameworks JUnit and Log4j. We also show that the detected hotspots of Log4j and JUnit are consistent with their respective documentations.

Categories and Subject Descriptors: D.2.3 [Software Engineering]: Coding Tools and Techniques—*Object-oriented programming*; D.2.6 [Software Engineering]: Programming Environments—*Integrated environments*;

General Terms: Languages, Experimentation.

Keywords: Hotspots, Code search engine, Code reuse

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 often offer complex procedures 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.

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*. The foundations of hotspots are built upon the Open-Closed principle by Martin [2]. 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 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 that reuses 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 improvement) on these hotspots because the resulting returns on investment may be substantial.

Detecting hotspots of an input open source framework requires domain knowledge of how the APIs of the input framework are reused by applications, referred as client applications. On the web, various open source repositories are available and these open source repositories include versions of open source projects that reuse APIs of the input framework. These open source repositories can serve as a basis for gathering the information of how APIs of the input framework are reused, and hence can help in detecting hotspots. Therefore, our approach, called SpotWeb, leverages a code search engine (CSE) to gather related code examples of APIs of the input framework from these open source repositories. Given a query, a CSE can extract code examples with usages of the query from open source repositories available on the web. Our approach analyzes gathered code examples statically and detects hotspots of the given input framework. Our approach is the first approach that extends the scope of client application code bases for detecting hotspots to billions of lines of code in open source repositories by leveraging a code search engine. Our approach tries to address the

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

MSR'08, May 10-11, 2008, Leipzig, Germany.

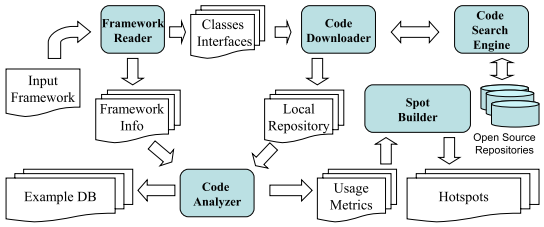

Figure 1: Overview of the SpotWeb approach

problems related to the quality of the code examples gathered from a CSE by capturing the most common usages of the APIs through mining. We use our SpotWeb approach to detect hotspots of two widely used open source frameworks: JUnit and Log4j. We show that the detected hotspots are consistent with the starting points described in the documentation for these frameworks.

2. APPROACH

Our approach consists of five major components: the framework reader, CSE, code downloader, code analyzer, and spot builder. Figure 1 shows an overview of all components and flows among different components. We use JUnit as an illustrative example for explaining our approach.

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.

Code Downloader. The code downloader interacts with a CSE to download related code examples. For example, the code downloader constructs a query such as "`lang:java junit.framework.TestSuite`" for gathering related code examples of the `TestSuite` class. The downloaded code samples, referred as *LocalRepository*, are given as input to the code analyzer. The code samples stored in the *LocalRepository* are partial as CSE gathers individual source files, instead of entire projects.

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. As these code samples are partial, the code analyzer uses several type heuristics for resolving object types. These type heuristics are described in our previous approach, called PARSEWeb [4]. The *UsageMetrics* capture several ways of how often a class or an interface or a method of the input framework is used by the gathered code samples. 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,

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 ($UM_i \neq 0$)
    if (Position of $MET_i \leq$ (HT * Size of $SortedMET$))
      Set MET_i type as $HOTSPOT$;
    else Set MET_i type as $WEAKSPOT$;
}
{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
for (C_i in {C_1, ... C_n}) {
  Rank of C_i = Minimum rank among all MET_i of the C_i;
  switch (C_i) {
    case (Interface): Set type of C_i to $HOOK$;
    case (Abstract Class): Set type of C_i to $HOOK$;
    other: {
      if (EX_i of $C_i > IM_i$ of C_i) Set type of C_i to $HOOK$;
      else Set type of C_i to $TEMPLATE$;}}
Group C_i of the same type into hierarchies based on inheritance;
Associate hook hierarchies to template hierarchies;
Define dependencies between template hierarchies;
Output hook and template hierarchies as hotspot hierarchies;

Figure 2: 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*/
}
```

```
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*/
}
```

```
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*/
}
```

Figure 3: Example classes of a sample framework

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 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 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.

Spot Builder. The spot builder component (SBC) uses computed *UsageMetrics* for detecting hotspots. The algorithm used by SBC for detecting hotspots is shown in Figure 2. We next describe the algorithm through an illustrative example shown in Figure 3. 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 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 hotspot methods. For example, for a HT of 45%, SBC identifies the methods such as m_{3_1}, m_{3_2}, m_{3_3}, and c_1 as hotspot

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

Figure 4: Recommended code example for *hook*

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

Figure 5: Recommended code example for *template*

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 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 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 group 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 the arguments passed to methods of the 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 accept 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.

We developed SpotWeb as an Eclipse plugin. SpotWeb can be invoked by selecting a menu item available on projects in Eclipse. In the SpotWeb implementation, we used the HT percentage of 25%.

3. EVALUATION

We evaluated SpotWeb with JUnit and Log4j frameworks to show that our approach can detect hotspots described in their respective documentations. We also explain how

Subject	# Classes	Hotspots				
		# Classes	%	# Templ	# Hooks	# Depend
JUnit	56	23	41.07	11	5	11
Log4j	207	74	35.74	49	13	44

Table 1: Evaluation results with JUnit and Log4j

SpotWeb results can be used by a framework user. The primary reason for selecting Log4j[1] and JUnit[2] for analysis is the availability of their documentation that can help validate the detected hotspots.

Table 1 shows the number of hotspots detected in each framework. The table also shows the number of templates, hooks, and the number of dependencies among templates and hooks. SpotWeb detected all hotspots described in documentations of JUnit and Log4j resulting in a recall of 100%. SpotWeb detected a few other hotspots that are not described in documentation, resulting in a precision of 26.08% for JUnit and 16.21% for Log4j.

We next explain hotspots detected in the JUnit framework. Figure 6 shows the hotspot hierarchies detected in the JUnit framework. 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 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). SpotWeb tries to capture the usage relationships among hotspot classes through dependencies. 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 6, consider that a framework user wants to start with the method `addTest` of the template class `TestSuite` in Hierarchy 5. Figure 6 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 4 for the hook class `TestCase`, which is a part of Hierarchy 1.

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

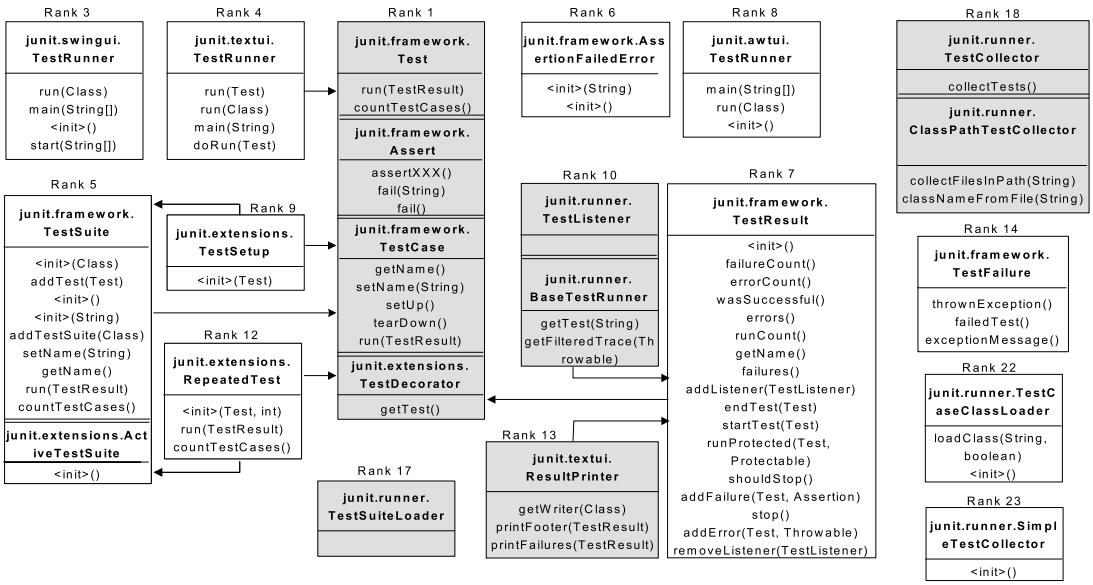

Figure 6: Hotspots identified for the JUnit framework

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 of `test`. SpotWeb also recommends a code example for the `addTest` method and the recommended code example is shown in Figure 5. 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.

We next describe the detected hotspots in the Log4j framework. Log4j provides several features such as Appenders and Layouts, and for each such feature, Log4j provides several classes. Among those several classes, a few classes are much more often used than other classes. SpotWeb correctly detected the classes that are often used for each such feature. For example, SpotWeb detected that `ConsoleAppender` and `FileAppender` are the commonly used classes for the Appender feature. 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`.

4. RELATED WORK

Mendonca et al. [3] 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 [1] 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. Therfore, 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 extends the data scope by leveraging a CSE.

We used CSEs for gathering related code examples in our previous approaches called PARSEWeb [4] and MAPO [5]. However, these previous approaches are developed for assisting the users in effectively reusing a given API. In contrast, SpotWeb assists framework users by detecting hotspots that can serve as starting points for reusing the framework.

5. CONCLUSION

In this paper, we proposed an approach called SpotWeb that assists software developers in reusing APIs of an existing framework by detecting hotspots of the framework. These hotspots can serve as starting points for reusing the framework. SpotWeb addresses 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 two open source frameworks and showed that the detected hotspots are consistent with the starting points described in the documentation for these frameworks.

6. REFERENCES

[1] R. Holmes and R. J. Walker. Informing eclipse api production and consumption. In *Proc. ETX*, pages 70–74, 2007.

[2] R. C. Martin. The open-closed principle. In *More C++ gems*, pages 97–112, 2000.

[3] 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.

[4] S. Thummalapenta and T. Xie. PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proc. ASE*, 2007.

[5] T. Xie and J. Pei. MAPO: Mining API usages from open source repositories. In *Proc. MSR*, pages 54–57, 2006.