Submitted for consideration to GPCE 2015

Native-2-Native

Automated Cross-Platform Code Synthesis from Web-Based Programming Resources

Antuan Byalik, Sanchit Chadha, and Eli Tilevich

Department of Computer Science
Virginia Tech, Blacksburg, VA 24061, USA
{antuanb,schadha,tilevich}@cs.vt.edu

Abstract

For maximal market penetration, popular mobile applications are typically supported on all major platforms, including Android and iOS. Despite the vast differences in the look-and-feel of major mobile platforms, applications running on these platforms in essence provide the same core functionality. As an application is maintained and evolved, the resulting changes must be replicated on all the supported platforms, a tedious and error-prone programming process. Existing automated source-to-source translation tools prove inadequate due to the structural and idiomatic differences in how functionalities are expressed across major platforms.

In this paper, we present a new approach—*Native-2-Native*—that automatically synthesizes code for a mobile application to make use of native resources on one platform, based on the equivalent program transformations performed on another platform. First, the programmer modifies a mobile application's Android version to make use of some native resource, with a plugin capturing code changes. Based on the changes, the system then parameterizes a web search query over popular programming resources (e.g., Google Code, StackOverflow, etc.), to discover equivalent iOS code blocks with the closest similarity to the programmer-written Android code. The discovered iOS code block is then presented to the programmer as an automatically synthesized Swift source file to further fine-tune and subsequently integrate in the mobile application's iOS version. Our evaluation, enhancing mobile applications to make use of common native resources, shows that the presented approach can correctly synthesize more than 86% of Swift code for the subject applications' iOS versions.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; D.2.7 [*Software Engineering*]: Portability

General Terms Algorithms, Experimentation

Keywords Recommendation Systems, Code Synthesis, Mobile Computing, Android, iOS, Java, Swift

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

The mobile application market remains fragmented, with several major platforms, including Android, iOS, and Windows Phone, competing to dominate market share. Nevertheless, shrewd mobile software vendors commonly support their popular applications on all major platforms to maximize their customer base. On each supported platform, an application replica essentially delivers an identical set of functionality, albeit within the conventions and formats of the platform at hand. For example, a mobile application with a map component would use Google Maps on Android and Apple Maps on iOS Devices. Even though from the end-user's perspective, the provided map feature provides identical functionality on both platforms, from the software engineering perspective, implementing the same feature on different platforms requires the use of vastly dissimilar languages, APIs, and software architectures. For instance, Android applications are written in Java using the Android standard library, in which UI events are expressed by means of callbacks; meanwhile, iOS applications are written in Swift using the iOS standard library, in which UI events are expressed by means of delegates.

As a mobile application evolves with new features and functionalities, the mobile developers must replicate the changes on all supported platforms. Having overcome the challenges of ascertaining the correct program logic and implementation details on one platform, the developer has no choice but to repeat the same tedious and error-prone process on all the remaining supported platforms. In other words, the developer is unable to leverage the expertise gained by undertaking a programming task on one platform to facilitate the performance of that same task on other platforms. What if it were possible to automatically glean the knowledge acquired by adding a feature to an application on a source platform to semi-automatically synthesize the code required to add the same feature on target platforms?

In this paper, we present `native-2-native`—a novel approach that applies a code synthesis algorithm to discover publicly available Swift code blocks whose semantics are equivalent to a Java code block written for the Android platform. The approach starts with the programmer adding a feature to an application's Android version. The approach then logs the manually written Java/Android code and uses it to search the web for the available Swift/iOS code that implements the same functionality. A ranking algorithm applies a high-dimensional feature vector to select the code block whose functionality is the closest to the original Java/Android code. The selected code then parameterizes a code generator that synthesizes a semantically correct Swift source file that can be included into the the application's iOS version, requiring minimal manual fine-tuning in most cases. Our approach focuses on mobile applications' native resources, such as sensors and services. Because the majority of mobile applications nowadays need to make use of such

native resources, our approach aims at facilitating one of the most common programming tasks undertaken by the modern mobile application developer. It is the ability to automatically discover a code block that natively implements a feature in Swift/iOS for an equivalent code block implemented natively in Java/Android that gives our approach the name of `native-2-native`.

Our reference implementation of `native-2-native` makes use of the Eclipse IDE to provide a plugin for the Android development environment. The plugin captures and analyzes the token frequency in the Java code block that accesses a resource by means of some native API. Based on the captured code, the plugin forms a metadata query to search popular Web-based programming resources for Swift code blocks accessing equivalent native APIs on the iOS platform. The highest ranked discovered Swift code blocks are provided to the developer who can further refine them with extra functionality, such as fault tolerance capabilities or strengthened security. Our evaluation shows that the automatically discovered Swift code for accessing subject native APIs ends up fit as is for the task at hand in more than 86% of test cases. These results indicate that the presented approach can become a pragmatic and powerful tool in the toolset of developers charged with the challenges of supporting mobile applications on multiple platforms.

The rest of this paper is organized as follows. Section 2 presents a running example. Section 3 describes the approach Section 4 details our evaluation and discusses the strengths and limitations of our approach. Section 5 compares the presented approach with the related state of the art. Section 6 presents future work directions and conclusions.

2. Running Example

Next we provide a concrete example that motivates the need for `native-2-native`. Consider a mobile application that determines if any of the user's friends are in the vicinity. Hence, the application is in essence a person proximity locator that continuously retrieves and processes GPS location information from the requesting device. Let us assume that the application is supported on both Android and iOS.

2.1 Obtaining GPS Location Information

The left part of Figure 1 shows the code block that retrieves GPS location in Android; the right part shows equivalent functionality in iOS. Even though both code blocks accomplish the same task, they are structured quite dissimilarly. In particular, the Android version creates a `locationManager` object from the passed context object, so that the initialized object can be queried for the GPS information. The iOS version creates a `CLLocationManager` object, whose initialized state contains the latitude and longitude variables. The code blocks on both platforms are relatively short, following similar coding idioms (i.e., creating an object to query its state) to retrieve the GPS information. Nevertheless, having written the code block in Android would not equip the programmer with the required knowledge to replicate this basic functionality in Swift.

Despite the popularity of source-to-source translators, they would be inapplicable if one wanted to automatically derive the iOS version of the code. The reason for the ineffectiveness of source-to-source translation in this instance is that native API access is always domain-specific, a complication that cannot be tamed with syntax-directed translation. By contrast, `native-2-native` extracts domain-specific semantics from the Android code block to be able to search for equivalent Swift code. By automating the processes of extracting a code block's semantics and of discovering existing Swift examples with the closest equivalent functionality, the presented approach can alleviate some of the most tedious and error-prone programming tasks in modern mobile development.

2.2 Enabling Insights

Heretofore, the discussion in this paper has focused on the *what* component of our approach—our end goal of automatically synthesizing native code for a target platform from equivalent code on a source platform, thereby enabling a cross-platform translation of natively implemented features. Before explicating the *how* component, which will provide our approach's algorithmic and implementation details, next we will focus on the *why* component, which will identify our approach's enabling insights.

The presented approach is enabled by a confluence of the following insights, derived from observing the realities of modern mobile software development: the peculiarities of the mobile software market, the working preferences of the modern mobile programmer, and the nature of platform-specific mobile APIs. We next describe each of these insights in turn.

Major mobile platforms have been competing with each other for market domination. Mobile hardware vendors have embraced the competitive mindset, which results in a so-called "arms race" when it comes to the devices' features and capabilities. As soon as one platform introduces a new hardware enhancement, the competitors feel compelled to introduce the equivalent or improved enhancement on their platform, lest they were to lose a major marketing advantage. Consider the GPS sensor, which provides the foundation for our running example. If this sensor and its corresponding capabilities were to be introduced to the Android platform first, the iOS platform would have no choice not only to mimic this feature, but also to add extra enhancements in the closest release feasible. Subsequently, Android would be compelled to match the latest iOS progress in this area without delay. This continuous competitive cycle, although driven exclusively by market forces, unveils the first enabling insight of our approach: major mobile platforms necessarily share an excessive amount of core native features.

Although their implementation languages and the corresponding native API's may differ vastly, their underlying feature sets from the end user's perspective enjoy a remarkable degree of similarity, which in turn leads to a high level of correlation in the vocabulary used to express the native APIs for these corresponding features. Assuming that API designers aim at creating intuitive-sounding and easy-to-understand names, reading in GPS information can be expressed in a finite, reasonably sized number of ways. It is also highly likely that the ways the GPS APIs are expressed on different mobile platforms will overlap in non-insignificant ways. Going back to the code blocks for this feature in Figure 1, one can see that the tokens *location* and *location manager* are both heavily used in both Java/Android and Swift/iOS. It is these shared tokens that make it possible to design a cross-platform translation mechanism for native APIs for shared features. Furthermore, the number of analogous features and their corresponding API shared tokens will continue increasing unless some platform definitively wins the competition for market share.

Based on the growing number of online programming resources, the modern programmer increasingly relies on the Web to answer questions that come up during their day-to-day operations ranging from simple bug fixes to complex refactoring. For example, StackOverflow [17] receives 2.65 new questions per minute and 4.41 new answers per minute on average, reaching close to 6,000 questions a day in peak sessions [18]. Indeed, StackOverflow and similar online question/answer discussion forums have become the modern programmer's primary medium of information exchange. One can attribute this strong shift toward online programming documentation to sheer demographics—StackOverflow reports that the average age of their user is 28.9 year old, based on surveying over 26K developers across 157 countries [19], which places them strongly within Generation Y, also known as the first digital generation, used to rely on the Web for all kinds of infor-

```java
public static LocationModel getLocation(Context context) {
    locationManager = (LocationManager)context.
      getSystemService(Context.LOCATION_SERVICE);
    Criteria criteria = new Criteria();
    String bestLocation = locationManager
      .getBestProvider(criteria, false);
    Location location = locationManager.
      getLastKnownLocation(bestLocation);
    LocationListener loc_listener = new LocationListener()
    {
        public void onLocationChanged(Location l) {}
        public void onProviderEnabled(String p) {}
        public void onProviderDisabled(String p) {}
        public void onStatusChanged(String p, int status,
          Bundle extras) {}
    };
    locationManager.requestLocationUpdates
      (bestLocation,0,0,loc_listener);
    location = locationManager.
      getLastKnownLocation(bestLocation);
    LocationModel loc = new LocationModel
      (location.getLatitude(),location.getLongitude());
    return loc;
}
```

```swift
func startLocationUpdate() {
    locManager.requestWhenInUseAuthorization()
    locManager.startUpdatingLocation()
}

func locationManager(manager: CLLocationManager!,
  didUpdateLocations locations: [AnyObject]!) {

    var location = locations.last as! CLLocation
    var lat:Double = location.coordinate.latitude as
      Double
    var long:Double = location.coordinate.longitude
      as Double
    var result = NSString(format: "%.5f, %.5f",
      location.coordinate.latitude,
      location.coordinate.longitude)
    self.location = result as String;
}

func locationManager(manager: CLLocationManager!,
  didFailWithError error: NSError!) {

    DLog("Location Error: " + error.description);
}
```

Figure 1. Android (left) and iOS (right) get Location basic functionality

mation. At any rate, it is indisputable that the Web has become an invaluable information sharing and acquisition platform for mobile developers. An additional draw of programming resources web sites is serving as reputation builders. Users of these websites commonly have the ability to rank the quality of provided information, with top providers earning high degrees of prestige and notoriety. These digital rankings can also be leveraged to automatically assess the quality of the available programming information.

Finally, we argue that figuring out how to express a native API on some platform, having just expressed an equivalent API on another platform, constitutes *accidental complexity*, and as such is a promising candidate for automated treatment via software engineering innovation [4]. Consider the process by which some native API becomes used in a typical mobile program. A developer decides that some feature needs to be added to a program, and that feature will make use of some native resource. Designing the feature is *essentially* complex, while discovering which API one must invoke is *accidentally* complex. Furthermore, this discovery process remains accidentally complex, irrespective of the implementation platform. Removing accidental complexity is a key driving force behind reducing the programmer workload. There is another peculiarity that arises when translating native APIs between Java and Swift. While Java remains the most commonly used programming language [19, 22], Swift according to StackOverflow is now regarded as "most loved" language. Hence, one can expect an abundance of web-based programming resources for both languages.

By leveraging these three main insights, we were able to create a simple but powerful approach to automatically translating native APIs between Java/Android and Swift/iOS. In the next section, we provide the algorithmic and implementation details of our approach. In Section 4, we report on the results of applying our approach to real-world examples of using native APIs.

3. The Native-2-Native Approach

Figure 2 shows a high-level overview of our approach. A mobile application developer first implements a new feature using some native API of the Android platform in Java. Once the developer deems the feature's implementation completed, the feature's code block is passed as input into the `Native-2-Native` IDE plugin. The plugin performs the following tasks in sequence: generalize the Java code block to form a search query, execute the query against popular web-based programming resources, summarize and rank the results, and finally present a synthesized Swift code block for the iOS platform back to the developer. The presented code block is typically partially complete, and implements the same feature as the input Java code, but by means of the equivalent native Swift/iOS API.

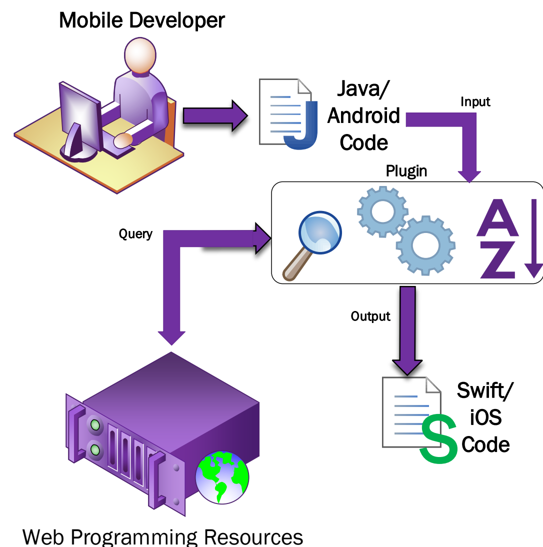

Figure 2. Native-2-Native: High-level Approach Overview

In the following sections, we detail the constituent parts of the native-2-native approach. Section 3.2 describes the process of extracting core functionality from the input Android/Java code block. This process includes multiple techniques in tokenization, filtra-

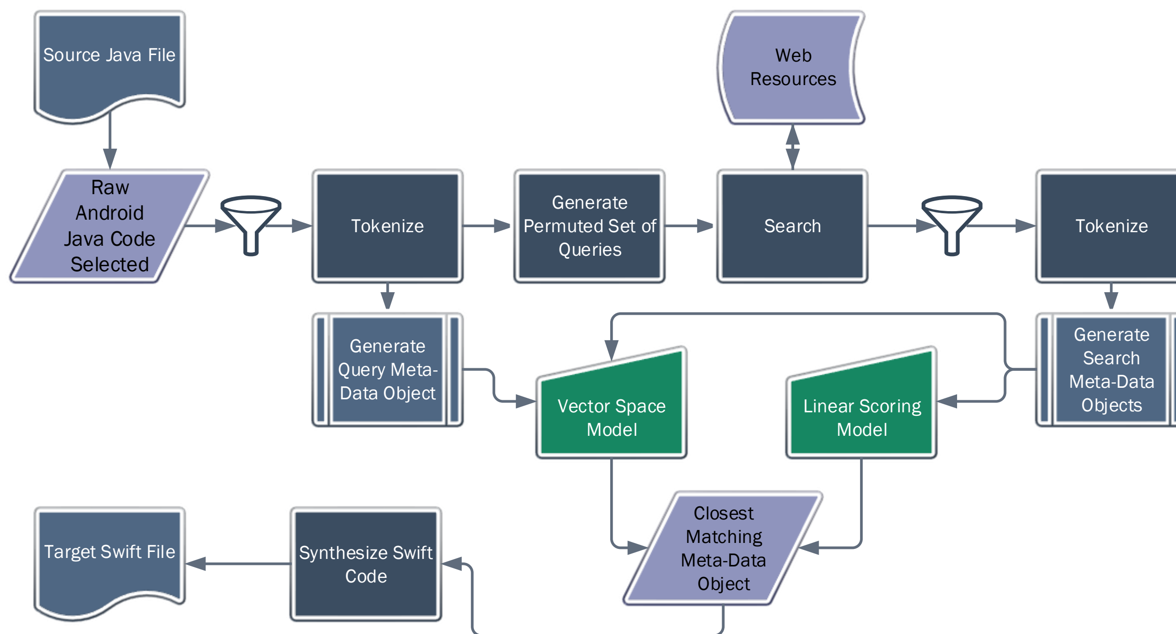

Figure 3. Full approach program flow

tion, and frequency analysis to generate an important set of query keywords. Next, section 3.3 explains how the approach ascertains a similarity index between the initial Java code block and all mined Swift code blocks. The meta-data objects serve as the central representation of platform and language-independent semantics for all mined documents as well as the input Java source code. Finally, section 3.4 delves into the underlying process by which the approach is able to produce an equivalent Swift code block. This process makes use of two algorithms: (1) a searching algorithm that discovers the relevant resulting set of documents, and (2) a ranking algorithm that operates on a set of mined documents to produce a platform- and language-independent semantic ranking, which synthesizes the output Swift code block.

3.1 Terminology

For the rest of the presentation, the term *source document* will refer to the input Android Java code block, while *target document* will refer to the output Swift code block. *Query* keywords will refer to the extracted core functionality from the input Android code block, used to search for the target document's constituent components. Finally, *token* will refer to any single document element at the level of individual strings.

3.2 Extracting Core Functionality from Java/Android Code

This section describes the process of translating input Android Java code blocks into web queries. This process extracts the core functionality of the code block by means of tokenization, filtering, and frequency analysis, described in turn next.

3.2.1 Tokenizing and Filtering

The flowchart in Figure 3 details the process. The Java source input is tokenized, with the superfluous tokens filtered out. The tokenizing makes use of the canonical bag-of-words model [8]. This model also separates camel case variables, title case variables, method declarations/invocations, and also removes non-alphanumeric characters attached to strings.

The resulting tokens are then first filtered by applying stemming in the form of suffix stripping to eliminate any verb-tense discrepancies that could arise while searching web resources for the target document's constituent components. Then tokens that happen to be substrings of other tokens are filtered as well. For example, tokens 'location' and 'loc' are assumed to possess related semantic intent. Tokens that would weaken the precision of the frequency analysis are filtered out as well (e.g., comment designators, stop words, etc.)

3.2.2 Frequency Analysis to Generate Web Queries

The next step calculates the document frequency for each extracted token that has not been filtered. The frequency analysis produces a sorted list of the most used tokens in the input. The k most used tokens become the query keywords for the search routine in Figure 4. The default value of k is 3 but can be customized at will.

3.3 Meta-Data Objects

Meta-data, data that describes other data, has been used to facilitate the search and discovery of related data objects [3, 16]. The presented approach uses meta-data to describe source and target documents as a means of streamlining similarity comparison. The source document's meta-date object is composed upon the completion of the tokenizing and filtering steps as described next.

3.3.1 Meta-Data Fields

Meta-data fields serve as the abstraction that captures all pertinent information from web-based programming resources and the source document in an easily search-able and comparable format. The meta-data objects use their fields to store features mined from all the searched web resources as well as those extracted from the source document. A `generate` routine processes every search result to populate the fields defined by a given meta-data object. Rep-

resenting the search results via meta-data objects makes them easily amenable to similarity evaluation with various ranking models.

Some features are deliberately supplemented to indicate the functionality to search for. For example, a preferred StackOverflow response would be a so-called "accepted answer," a code snippet check-marked by its originator as having solved the posed question.[1] In contrast, the source document lacks this property, as it simply represents the input Android code block. To prevent supplemented features from unduly skewing the final ranking, the source document's missing features are defaulted in the meta-data object as their ideal functionality. Similarly, web search results returned from random programming resources will also contain missing features, as well their own supplemental features, to be steered toward the searched for functionality. The meta-data objects' fields are the union of all features across all mined web resources and the source document.

3.4 Searching and Ranking

In this section, we first present a searching algorithm that queries web-based programming resources for relevant data pertaining to the query keywords. Then, we present a ranking algorithm that incorporates two different models for determining similarity between the source document and all mined potential target documents.

It is worth pointing out that both the searching and ranking algorithms disregard the control flow constructs present in the source document, operating solely on the extracted keywords described in the previous section. This design decision renders our approach largely independent of the specifics of the business logic of the native code blocks at hand, focusing exclusively on the native API used. Nevertheless, the disregarded control flow constructs are fully restored during the final code synthesis phase.

3.4.1 Searching Algorithm

Figure 4 shows the searching algorithm for mining the relevant data. The algorithm's input is the generated query keywords discussed above and the output is a full list of the resulting searches stored in a custom-made, answer-wrapper object. Before the core of the searching algorithm is executed, the `initKeywords` subroutine first initializes the full query keyword set. Although the input is just the set of query keywords, various permutations and subsets of the original query keywords must be searched to locate all relevant results. The `initKeywords` subroutine in Figure 4 explains how the full set of keywords involves three components: 1) transcribe the original keywords as individual queries, 2) identify the first and second (by frequency-based importance) keywords, in both orders, as subset queries, and 3) permute the complete set of the original keywords. Although some flexibility in the number of query keywords is allowed for user fine-tuning, the hard limit of 5 separate keywords for the permutation component ensures that the input is computable in practical space and time boundaries.

Although the algorithm can be configured to search any web-based programming resources, we next describe it by means of three representative categories: (1) a live API call to StackOverflow with the given query keywords, (2) a Google search on the current query keywords that specifically limits to StackOverflow websites only to catch discrepancies in a StackOverflow search internally as well as an external Google search on the same keywords; the Google search results are funneled to StackOverflow to directly data-dump that post.[2] (3) a Google search that excludes StackOverflow posts to include less popular but potentially also relevant online blogs and other web resources to the list of relevant search results. All three methods' results are standardized into the answer-wrapper object and stored for later ranking and analysis. This entire process is conducted for each of the permuted list of query keywords to generate the final list of web-based search results.

[1] The StackOverflow website uses green check-marks to denote accepted answers.

[2] Searching through Google and StackOverflow frequently yields dissimilar complementary results in differing orders.

```
/* INPUT: query keywords to search */
/* OUTPUT: full list of resulting searches */
DEF Search(keywords)
    resultList ← ∅
    keywordSet ← initKeywords(keywords)
    FOREACH keyword ∈ ∀keywordSet DO
        queryResultsSOF = execStackOverflow(keyword)
        FOREACH result ∈ ∀queryResultsSOF DO
            Ans ← result, rank
            resultList ← resultList∪ {Ans}
        END FOREACH

        queryResultsGoogle = execGoogle(keyword)
        FOREACH result ∈ ∀queryResultsGoogle DO
            temp ← execStackOverflow(result)
            Ans ← temp, rank
            resultList ← resultList∪ {Ans}
        END FOREACH

        queryResultsElse = execElse(keyword)
        FOREACH result ∈ ∀queryResultsElse DO
            Ans ← result, rank
            resultList ← resultList∪ {Ans}
        END FOREACH
    END FOREACH
    RETURN resultList
END Search

DEF initKeywords(keywords)
    fullSet ← keywords
    fullSet ← fullSet ∪ {key[0] + key[1]}
    fullSet ← fullSet ∪ {key[1] + key[0]}
    fullSet ← fullSet ∪ perm(keywords)
    RETURN fullSet
END initKeywords
```

Figure 4. Search Algorithm Pseudo code

3.4.2 Ranking Algorithm

The ranking algorithm determines the degree of similarity between the source document and all potential target documents to present the most relevant Swift code blocks to the user. Figure 5 details the algorithm. The input is the list of search results generated by the searching algorithm, as well as the original source document (also in the answer-wrapper format), and a k value for the number of results to be returned. The output is the k-top results of the ranking. The algorithm's main components produce the vector space model and linear model with their respective feature sets. The standardization into answer-wrapper objects described above facilitates the retrieval of this information for each document in the results list. The standard format for the answer-wrapper objects is used to generate the target set of meta-data objects that produce their respective vector space and linear model scores. Next, the models are combined, sorted, and the top k of those are returned.

Figure 5 also includes the subroutine for calculating the `cosine` of the angle between the current potential target document and the source document, as required by the central logic of the vector space model [1]. The cosine determines similarity from a constructed n-dimensional sphere, containing all the n-dimensional feature vectors. Each vector's dimension is calculated using a term-frequency inverse-document-frequency (`tf-idf`) weighting function, which accounts for a term's frequency without over fitting by accounting for common terms across the document corpus, where

```
/* INPUT: source/targets MetaData, k top results */
/* OUTPUT: k closest ranked MetaDatas */
DEF Rank(Source, TargetMDs, k)
    ranking, topK ← ∅
    FOREACH target ∈ ∀TargetMDs DO
        C ← getCos(target, sourcelate)
        L ← getLin(target)
        temp ← bag(C, L)
        ranking ← ranking∪ (temp, target)
    END FOREACH
    sort(ranking)
    FOR i IN k DO
        topK ← topK ∪ ranking(i)
    END FOR
    RETURN topK
END Search

DEF getSimilarityUsingCos(t, s)
    N_t, N_s, cos ← ∅
    FOREACH d ∈ ∀t DO
        N_t ← N_t, d^2
    END FOREACH
    FOREACH d ∈ ∀s DO
        N_s ← N_s, d^2
    END FOREACH
    N_t ← sqrt(N_t)
    N_s ← sqrt(N_s)
    cos ← (t · s) / (N_t · N_s)
    RETURN cos
END getCos
```

Figure 5. Rank Algorithm Pseudo code

n is the total number of unique tokens available in the current document corpus. Each value in the vector is represented by the modified indicator function seen in equation 1 that incorporates the tf-idf weights. Here $d \in D$ represents some document in the full k set of documents, including all potential targets and the source, with weight w.

$$v_{d_k}(i) = \begin{cases} 0 & \text{if token } i \notin d_k \\ w_i & \text{if token } i \in d_k \end{cases} \tag{1}$$

The cosine value is calculated using the underlying vector space model shown in Equation 2 [16]. The norm of each vector is the square root of the summation of each dimension's squared value. Given that the dot product of two vectors is a scalar and this quantity is divided by the product of two norms, the resulting value is also a scalar. The variables s and t are used to denote the source and target documents, respectively.

$$cos_s(t) = \frac{s \cdot t}{||s||_2 \quad ||t||_2} \tag{2}$$

Note that the linear model subroutine is not shown in figure 5, as it is simply the linear combination of all relevant features in each meta-data object of the results documents shown in equation 3. The model's weights are derived from a combination of tf-idf values and normalization by average feature quantities available from the StackOverflow API.

$$\text{lin}(t) = \sum_{i=1}^{n} (w_i * t_i) \tag{3}$$

3.4.3 Complexity Analysis

In this section, we briefly comment on the computational complexity incurred by the more exhaustive procedures of the `native-2-native` approach. The ranking algorithm and following model combination process is the most computationally intensive component, on which we hence concentrate our analysis. Recall Equations 1 and 2, presented in the previous section, that outline the vector space model approach, as well as Equation 3 that explains the linear model used. Equation 4 shows the complete model combination, along with the weight calculation, as a function on all potential target documents. Note the parameters α and β that are constrained such that $\alpha + \beta = 1,\ \alpha \geq 0,\ \beta \geq 0$ and represent the respective magnitude of the two models to each other.

$$\max_{t \in T} \left\{ \frac{\alpha(s \cdot t_j)}{\sqrt{\sum_{i=1}^{n} s_i^2 \cdot \sum_{i=1}^{n} t_{i,j}^2}} + \frac{\beta \sum_{i=1}^{k} w_{i,j} f_{i,j}}{\max_{a \in A} \left(\sum_{i=1}^{k} w_{i,a} f_{i,a} \right)} \right\} \tag{4}$$

This equation is derived by combining the vector space model with a normalized linear model and using the weighting function shown in Equation 5. In Equation 5, note the $I(t)$ indicator function that is 0 if the current term is not present in the current document and 1 otherwise. Also of importance to this equation is T which represents the total number of target documents mined by the searching algorithm.

$$s, \forall t_j \in T : t_f \cdot \log \left(\frac{|T| + 1}{I(t)} \right) \tag{5}$$

Given that n_i is the total number of tokens in some document i, we bound the maximum number of tokens on the overall approach as $dim = \{\sum_{i=1}^{|T|+1} \{n_i\}\}$. This bound being placed in set notation to represent the elimination of duplicate tokens across not just a single document but all documents, and accounting for the full set T of target documents in addition to the single source document. Lastly, d is the upper bound on the feature vector used for the vector space model as we take all unique tokens. In essence, we have $|\langle d_{1,i}, d_{2,i}, ..., d_{n,i} \rangle| \leq dim$, where the vector represents some document d_i's n features; that will be equal across all targets and the single source document. Hence, the Big O of the approach's primary component in terms of documents is $\mathcal{O}(T+1)$, which runs in linear time, $\mathcal{O}(n)$. However, accounting only for documents fails to accurately reflect the true computational complexity, as for certain operations one must measure their more-numerous token operations to evaluate their complexity. Thus, the complexity measured in tokens is $\mathcal{O}(dim^2 + dim * d_i + dim * f + f)$, where f is the number of linear features, a number strictly smaller than dim, and where d_i is the current document's token count, also strictly smaller than dim as shown above. If $d_i \leq dim$, then $d_i * dim \leq dim^2$. Finally, we can simplify the overall complexity in terms of tokens as $\mathcal{O}(dim^2 + dim^2 + dim^2 + f) = \mathcal{O}(3dim^2) = \mathcal{O}(n^2)$.

3.5 Programming Interface and Code Synthesis

The approach is concretely realized as an Eclipse IDE plugin publicly available and open-sourced for future improvements and enhancements at `https://github.com/antuanb/Native-2-Native`. Figure 6 (left half) displays the initial plugin view from the end mobile developer's perspective. The developer first highlights a code block to be rendered in Swift and then clicks the generate button. Figure 6 (right half) highlights how the developer selects which of the top two presented results they desire to be presented as a Swift source file. Note the URL of the corresponding result is copied to the clipboard, so that the developer can refer to the originating web-resource for further information. The generated Swift source file is saved in the current working directory of the Android/Java project.

The left side of Figure 7 shows a sample generated Swift file in our running example of GPS location. The control flow of the Android/Java file is replicated to create a skeleton Swift source file

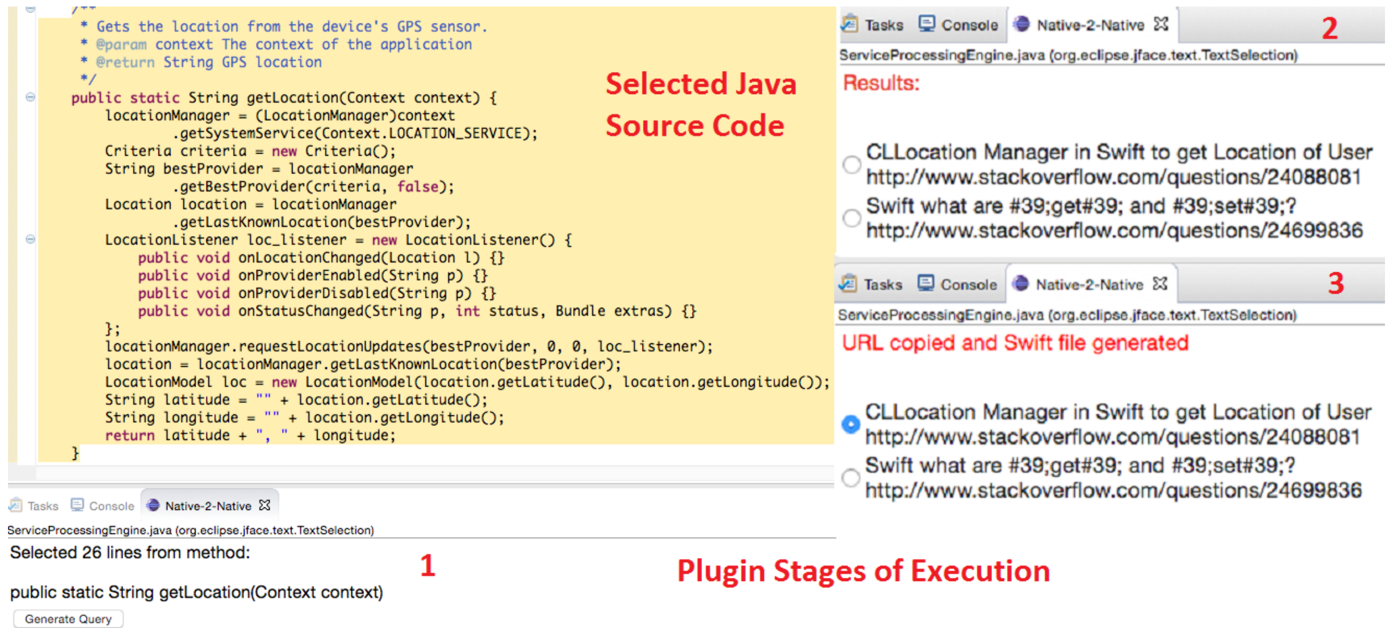

Figure 6. Plugin showing user selecting GPS Location code and subsequent results returned

first. Then, the developer-selected result is incorporated into the Swift file to finalize the synthesized code block. In this example, the synthesized Swift file has the correct iOS/Swift protocol for instantiating and utilizing the `CLLocationManager` native API to access a user's GPS location along with a logic flow outline. The remaining fine-tuning left for the developer is to format and return the contained location information in the style desired. The next section focuses on the evaluation of the presented approach, detailing the native APIs used, the evaluation process, and the precision levels obtained.

4. Evaluation

In this section, we present the results of evaluating our approach. To that end, we applied our approach to various Android/Java native APIs to automatically synthesize analogous functionality in iOS/Swift, and then examined the synthesized code blocks for their fitness in expressing the functionality at hand. We evaluated the Native APIs, including sensors (e.g., GPS, accelerometer, etc.), network interfaces (e.g., WiFi, Bluetooth Low Energy (BTLE), etc.), and canonical library classes/data structures (e.g., `String`, `ArrayList`, `HashMap`, etc.).

We evaluated all the synthesized functionality by hand, which included compiling the code with the Swift compiler and testing its runtime behavior. Future work will investigate whether this tedious evaluation process can be automated. The evaluation placed each automatically synthesized code block into one of the following four categories: (1) the synthesized code block appears to be correct both in form and functionality (i.e., it can be used carry out equivalent native API functionality when integrated with an iOS application); (2) the synthesized code block is incorrect, either in form or in functionality (i.e., the code cannot be compiled without major modification or its runtime functionality fails to deliver the expected equivalent native API functionality); (3) the synthesized code was partially correct, requiring a reasonable amount of programming effort to carry out the expected functionality. We define "reasonable programming effort" if the synthesized code block can be used as a helpful reference point that would speed up the search for the right functionality rather than hinder progress. Notice that our definition is necessarily subjective, thus creating an internal threat to validity. (Fortunately, our evaluation did not yield many results as belonging to this category.); (4) the result is not a code block, thus providing negligible utility to the programmer.

Table 1 displays the results of the aforementioned evaluation. The four evaluation categories for the synthesized code blocks are designated as `YES`, `NO`, `1/2`, and `NaCB` (not a code block), respectively. The `TOTAL` column reports on the total number of test cases for each native API type. Recall that our implementation furnishes the top two-ranked code block suggestions to the programmer. Then it is up to the programmer's purview to select the suggestion to be integrated in a given iOS application. The table lists the results that our approach produced as both `Rank 1` and `Rank 2`. The column `Rank 1 or 2` presents the `YES` results, which appeared in Rank 2 while missing in Rank 1; the relatively high number of cases in this column supports our design choice of the plugin furnishing the two top-ranked suggestions to the programmer.

The total evaluation results are summarized in the last two rows. Specifically, they report the total percentage of the `YES` outcomes obtained from Rank 1 only and from either Rank 1 or 2, respectively. While the first result is around 75%, the second one stands at more than 10 percentage points higher at almost 87%. One may wonder why we decided to limit our number of reported top suggestions only to two. This limitation is dictated by primarily practical considerations—for the majority of our test cases, considering more than two suggestions quickly proved impractical, taking additional time without a match in increased accuracy.

The `native-2-native` approach is not universally applicable and may fall short of the programmer's expectations in the presence of the impedance mismatch between the Java and Swift language vocabularies. The right side of Figure 7 shows one such example. The example makes use of the Java standard library's `HashMap` class, parameterized with `String` types as its key and value. Un-

```swift
//N2N-getLocation.swift
/**
* Gets the location from the device's GPS sensor.
* @param context The context of the application
* @return String GPS location
*/
func getLocation(/*unknown token*/) -> String {
    //Insert Native-2-Native result
    var latitude:String = "" /*unknownToken*/
    var longitude:String = "" /*unknownToken*/
    return latitude + ", " + longitude
}

/* NATIVE-2-NATIVE RESULT
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view,
       typically from a nib.
    locationManager = CLLocationManager()
    locationManager.delegate = self
    locationManager.desiredAccuracy = kCLLocationAccuracyBest
    locationManager.requestAlwaysAuthorization()
    locationManager.startUpdatingLocation()
}
*/
```

```swift
//N2N-putHashMap.swift
/**
* Associates the specified value with the specified key in this map. If the
* map previously contained a mapping for the key, the old value is
* replaced.
*/
func putHashMap(/*unknown token*/, B: String, C: String) -> /*unknown token*/ {
    //Insert Native-2-Native result
    var putHashMapTemp:/*unknown token*/ = A
    putHashMapTemp./*unknown token*/(B,C)
    return putHashMapTemp
}
/* NATIVE-2-NATIVE RESULT
extension String  {
    var md5: String! {
        let str = self.cStringUsingEncoding(NSUTF8StringEncoding)
        let strLen = CC_LONG(self.lengthOfBytesUsingEncoding(NSUTF8StringEncoding))
        let digestLen = Int(CC_MD5_DIGEST_LENGTH)
        let result = UnsafeMutablePointer<CUnsignedChar>.alloc(digestLen)
        CC_MD5(str!, strLen, result)
        var hash = NSMutableString()
        for i in 0..<digestLen {
        hash.appendFormat("%02x", result[i])
    }

    result.dealloc(digestLen)
    return String(format: hash)
    }
}
*/
```

Figure 7. Synthesized Swift code snippet for GPS Location (left) and HashMap (right)

		Rank 1				Rank 2				Rank 1 or 2
Native APIs	TOTAL	YES	NO	1/2	NaCB	YES	NO	1/2	NaCB	
String	25	19	4	2	0	12	4	8	1	22
ArrayList	22	13	5	3	1	6	5	1	10	18
HashMap/Dictionary	13	10	3	0	0	3	2	3	5	11
GPS Location	5	4	0	1	0	3	1	1	0	4
Accelerometer	2	2	0	0	0	1	0	1	0	2
BTLE	5	4	1	0	0	3	1	1	0	4
Wifi	5	4	1	0	0	4	1	0	0	4
Overall	75	**56**	14	6	1	32	14	15	16	**65**
Overall Yes (Rank 1 Only)		**74.7%**	**78.9% (Normalized)**							
Overall Yes (Rank 1 or 2)		**86.7%**	**84.9% (Normalized)**							

Table 1. Evaluation results for target native API code block synthesis as {yes, no, 1/2 suitable, NaCB (not a code block).}

fortunately, `HashMap` is a Java-specific concept that has very different names in other languages. In Swift, this programming idiom is supported by the `Dictionary` class. Without special provisions, `native-2-native` would return the code block depicted on the right side of Figure 7. This code block contains a use case of the MD5 hashing algorithm in Objective-C embedded into the control flow of the source document's Java code. This outcome would be a direct consequence of the extracted keywords (mapping, hash, and put) steering the search toward a hashing algorithm instead of a `dictionary` data structure. Because such vocabulary dissimilarities are inevitable, `native-2-native` special-cases several highly common instances of these dissimilarities by consulting a static mapping of vocabulary keywords between Java and Swift.

As a presentation choice, the final two rows of Table 1 treat every subject case first as equally important without normalizing across them, and then display the normalized counterparts. We argue that our presentation choice is well-founded. When it comes to needing assistance when supporting a piece of functionality on several platforms, the average mobile developer would likely care more about sensor APIs, such as `GPS` and `BTLE`, than fundamental data structures APIs, such as `String`, `ArrayList`, and `HashMap`.

Based on the results of our evaluation, one can conclude that the approach is effective enough to serve as a practical tool for mobile programmers supporting cross-platform applications. Because the automatically suggested code does not need to be perfect to provide a high degree of utility to the programmer, our algorithms proved surprisingly fit for the purposes intended. However, it is not only the fitness of our algorithms that explains the effectiveness of our approach. These algorithms work hand-in-hand with the realities of the mobile market, the availability of web-based programming resources, and the process of suggesting equivalent native APIs being manageable via tool automation.

Despite its practical utility, our approach has several limitations. The model underlying the searching and ranking algorithms of our approach is bound by the dimensions of the feature vector. In other words, the accuracy of the algorithms is reversely proportional to the size of the total number of unique tokens comprising a given code block. As a result, mobile developers are likely to find our approach most effective in those cases when they need to find equiv-

alent iOS code for small to medium (10-30 lines of code) Android native code blocks. The second limitation stems from the original closed development model for the iOS platform. Closed models traditionally result in reduced sharing of programming solutions. Exacerbating the conditions for evaluating the applicability of our reference implementation is a relative newness of the Swift language. In fact, we were surprised that our reference implementation was able to synthesize correct suggestions from a relatively limited set of web-based Swift programming resources. Nevertheless, several major technological trends are likely to address this limitation. For one, Swift will be open-sourced in coming releases, while the amount of available Swift code examples on the web seems to grow by leaps and bounds.

Finally, synthesizing Swift from given Java input is a necessarily difficult case of cross-language translation. Because Swift is much more declarative (i.e., concise) than Java, the translation must produce more declarative output from more loquacious input. As software engineering is becoming more declarative in terms of languages, specifications, and invariants, our approach holds a lot of promise for automatically transitioning current mainstream methods of expressing programming information into their declarative counterparts. For example, our approach can be used to get rid of the wordiness of anonymous inner classes in the pre-Java8 world, replacing this with code lambda expressions.

5. Related Work

`Native-2-native` is representative of a broad class of software engineering applications known as recommendation systems [14, 15]. Several examples of recommendation systems synthesize code snippets from web-based programming resources [10–12, 23] or build an intelligent code search engine [9]. We will discuss how our approach differs from or improves over these examples.

Prompter [12] is an Eclipse plug-in that given the current working code context automatically identifies relevant StackOverflow discussions. Its uniqueness is in providing a user-controlled confidence threshold to suggest only discussions that surpass this threshold. Compared to `native-2-native`, Prompter also makes use of the StackOverflow API, albeit as the only source for relevant discussion and code snippets. Our approach's larger search space [21], which includes Google Search, Google Code, and third-party resources in `native-2-native`), can achieve the level of precision required for cross-language and platform translation, a feature not supported by Prompter [12] or [23].

Some related approaches make use of statically cached programming resources to accelerate data retrieval. For example, the approaches presented in [11, 12] rank output code blocks by normalizing a sigmoid function of the average StackOverflow vote count from a June 2013 static data dump. Selene [20] recommends equivalent code blocks by searching a repository of 2 million example programs to provide usage examples for a given input code block. Sourcerer [2] is another code search engine for a large-scale code repository (SourceForge). Strathcona [6], similarly to the systems above, also uses a repository based search corpus and provides the user with a structural overview of relevant code rather than actual code examples and discussions. A recommendation system developed by Bacchelli et al. [1] utilizes a vector space model with *tf-idf* as its frequency weighting model along with a singular query corpus source of StackOverflow. Similarly to other systems, Seahawk [10] also uses a static and publicly available dump of StackOverflow questions and lacks support for Swift.

As mentioned above, StackOverflow receives close to 6000 new questions a day. A static data snapshot from 2 years ago may be sufficient to mine for information about established language ecosystems and environments. However, the focus of `native-2-native` is mobile computing with rapidly evolving programming environments and language ecosystems. By combining vector space and linear models on live StackOverflow data, `native-2-native` returns suggestions that are more relevant, up-to-date, and better geared toward newer languages, such as Swift. Seahawk motivates the necessity of using the static dump to be able to search by means of the *Apache Solr* search system to achieve high performance efficiency. Although it would be unrealistic trying to match the performance efficiency of searching against a static local snapshot, we discovered that carefully calibrating weights and feature sets for the *tf-idf* analysis and vector space model not only provides complete and relevant results for both StackOverflow posts and other code sources, but also yields performance levels sufficient for practical use. Lastly, [9] focuses on providing useful documentation that supersedes standard API usage documentation. While an important aspect of the developer's ability to create mobile applications can at times include understanding necessary documentation, `native-2-native` focuses instead on the code synthesis and not just on supplemental documentation for the developer.

The recentness of the Swift language's entry into the mobile computing space renders mainstream native transpilation (i.e., source-to-source compilation) systems inapplicable. For example, Google's J2ObjC [7] converts pure Java source code into Objective-C source code. Although a powerful and practical tool used by Google internally, J2ObjC lacks support for Android APIs and the controller component of the MVC design pattern, as well as converting to Objective-C rather than the new standard of Swift. By contrast, `Native-2-native` embraces the native mobile APIs such as Android and iOS. While it remains unclear whether rule-based compiler translation is even capable of bridging the differences between the platforms as architecturally dissimilar as Android and iOS, in the meantime `Native-2-native` provides a practical solution for deriving working Swift code analogous to its Android counterpart.

`Native-2-native` utilizes common themes and features across various prior recommendation systems, but it applies them to a problem that arises from the realities of modern mobile development—the necessity of supporting popular mobile applications on all major platforms, despite the inherent dissimilarities in the platforms' languages, APIs, and architectures. Potentially, this problem may be addressed by state-of-the-art cross-platform source-to-source translation, with some inroads already in place [13]. Nevertheless, precise source-to-source compilation capable of translating Java Android to Swift iOS functionality for native APIs remains a futuristic vision.

By using sources other than user-driven StackOverflow posts, `Native-2-native` mines a wider feature net without being restricted to API mappings [24]. This feature enables our approach to provide potentially more valuable and relevant search results based on the developer's code context. Given that the overriding goal of `native-2-native` is to aid the developer by suggesting and synthesizing the iOS/Swift equivalent of Android/Java code, any new suggested Swift code, as long as it correctly implements some iOS API, is likely to prove useful to the mobile developer.

6. Future Work and Concluding Remarks

One potential direction for improving this work entails reusing the search data in an intelligent manner. Our current approach relies on generating a document corpus upon each query, but these results are discarded for all future queries, not just locally for a particular user but across all users. If instead, these meta-data objects were saved globally to become accessible for all users, then our approach can be further optimized in the following two major ways. First, the potential use of a preference server would facilitate an ability to predict the semantics of future queries made by a particular user based on previous inquiries they made. This predicted potential

query could be weighted along with the newly generated meta-data object and form the user's next query. Secondly, we could store all meta-data objects across all users in a set of online clusters. The clusters would be defined by some similarity measure related to the original ranking model. This optimization would facilitate a speedup when a user's query is within the cluster and within some threshold of similarity specified by the user, as one then would not need to continue a full web search for potential Swift code blocks.

Another potential avenue of improvement is to incorporate the translation of the "Model" component of an application's Model-View-Controller pattern. This enhancement could be easily accomplished by integrating with `native-2-native` the previously discussed [7] or another source-to-source translator of non-Android Java code. Similarly, we can further improve the translation and ranking by allowing users to rate our approach's returned Swift code blocks. These ratings would then be incorporated into future queries not just for this user but for all users making similar queries.

Yet another future work direction would expand the number of target platforms to Windows Phone and platform-independent JavaScript frameworks such as PhoneGap [5]. The developer would implement a new feature on one platform and then will get suggestions for all the other supported platforms. Comparing the software metrics of the equivalent code blocks on different platforms can shed insights on various software engineering properties of different languages and architectures.

This paper has presented a novel approach for automatic code synthesis from Android/Java to iOS/Swift by utilizing popular web-based programming resources. To enable our approach, we first gleaned several insights underlying the realities of modern mobile software development and the mobile computing market. Our approach is concretely realized as `native-2-native`, which includes, primarily extracting core functionality from input Java source code, searching, ranking, and code synthesis. The reference implementation of `native-2-native` is an Eclipse plugin that allows the developer to select input Java source code and chose from the top two returned search results before using this selection to synthesize the output Swift source file. The evaluation of our approach and its reference implementation show that `native-2-native` produces useful and intended functionality in as many as 86% of subject native APIs. These results indicate that the presented approach can become a pragmatic and valuable programming tool in the arsenal of mobile software developers.

Availability `native-2-native` is available from `https://github.com/antuanb/Native-2-Native`. The site includes the full source code for the approach, including the integration with Eclipse, open-source license, detailed results, and additional evaluation use cases.

Acknowledgments

This research is supported in part by the National Science Foundation through Grant CCF-1116565. We would like to thank Brendan Avent, Young-Woo Kwon, Alla Rozovskaya, and Myoungkyu Song for their insightful comments and suggestions for improving the manuscript.

References

[1] A. Bacchelli, L. Ponzanelli, and M. Lanza. Harnessing stack overflow for the IDE. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, RSSE '12, pages 26–30. IEEE Press, 2012.

[2] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 1–4. IEEE Computer Society, 2009. .

[3] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996. ISSN 0885-6125. .

[4] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, apr 1987.

[5] R. Ghatol and Y. Patel. *Beginning PhoneGap: Mobile Web Framework for JavaScript and HTML5*. 2012.

[6] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 117–125. ACM, 2005. .

[7] J2ObjC.org. J2objc, 2015.

[8] P. Jackson and I. Moulinier. *Natural language processing for online applications. Text retrieval, extraction and categorization*, volume 5 of *Natural Language Processing*. Benjamins, 2002.

[9] J. Kim, S. Lee, S.-w. Hwang, and S. Kim. Towards an intelligent code search engine. 2010.

[10] L. Ponzanelli, A. Bacchelli, and M. Lanza. Seahawk: Stack Overflow in the IDE. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1295–1298. IEEE, 2013.

[11] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 102–111. ACM, 2014. .

[12] L. Ponzanelli, G. Bavota, M. D. Penta, R. Oliveto, and M. Lanza. Prompter: A self-confident recommender system. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 577–580. IEEE Computer Society, 2014. .

[13] A. Puder and O. Antebi. Cross-compiling android applications to ios and windows phone 7. *Mob. Netw. Appl.*, 18(1):3–21, Feb. 2013. ISSN 1383-469X. .

[14] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *Software, IEEE*, 27(4):80–86, July 2010. .

[15] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, editors. *Recommendation Systems in Software Engineering*. Springer, 2014. .

[16] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, Nov. 1975. ISSN 0001-0782. .

[17] StackOverflow.com. Stack overflow, 2015.

[18] StackOverflow.com. Usage of /info - stack exchange api, 2015.

[19] StackOverflow.com. Stack overflow developer survey 2015, 2015.

[20] W. Takuya and H. Masuhara. A spontaneous code recommendation tool based on associative search. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, SUITE '11, pages 17–20, 2011. .

[21] B. Vasilescu, V. Filkov, and A. Serebrenik. Stackoverflow and github: Associations between software development and crowd-sourced knowledge. In *Social Computing (SocialCom), 2013 International Conference on*, pages 188–195, Sept 2013. .

[22] www.tiobe.com. Usage of tiobe language use statistics, 2015.

[23] A. Zagalsky, O. Barzilay, and A. Yehudai. Example overflow: Using social media for code recommendation. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pages 38–42, June 2012. .

[24] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining api mapping for language migration. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 195–204. ACM, 2010. .